Master-Thesis in Computer Science and Media

Fachbereich Druck und Medien der Hochschule der Medien, Stuttgart

Architecture and Prototype of a Game-Engine

Architektur und Prototyp einer Spiele-Engine

Andreas Stiegler

Examiner: Prof. Walter Kriha
Computer Science and Media
Stuttgart Media University

Prof. Dr. Jens-Uwe Hahn
Computer Science and Media
Stuttgart Media University

Stuttgart, November 16th, 2010

Stuttgart Media University

Hochschule der Medien, Stuttgart

"Imagination is more important than knowledge.
For knowledge is limited [...]."

"Phantasie ist wichtiger als Wissen,
denn Wissen ist begrenzt [...]."

- Albert Einstein, 1929

1. Abstract

This thesis describes the modules of a modern, commercial-scale game engine. It also gives an overview of potential techniques and implementations with examples of real commercial titles. The term "game engine" is defined, and the development of a game engine is sorted in the development process of a whole game project. The requirements of a game engine are identified in respect to the target genre or meta-genre and different models for managing a game engine are shown. Each subsector of a game engine, excluding rendering and AI, is then closely analyzed to identify how entities can be mapped on the game engine runtime. In contrast to many other documents, this thesis will not view a game engine from a "graphics" point of view, starting development at the graphics engine. Instead, this paper will lead to actually implementing an in-game universe, taking care of causality, time, and information in the process and then transferring the archived knowledge into an implementation.

After an overview of the functionalities of a commercial-scale game mechanics oriented game engine are established, this thesis will introduce an exemplary engine: the Telesto Engine, designed for real-time-Strategy Games and Massive Multiplayer Online Games (MMOGs). An overview of Telesto is given, forming a link to the generic engine descriptions. Likewise, the most important implementations of Telesto are uncovered, including the actual algorithms used for the time and causality handling, the scripting language, the resource manager, the physics and many smaller modules, such as the GUI or a generic extractor pattern to connect external modules like a graphics renderer.

1.1. About this document

This document is split into two parts. The first part establishes a common vocabulary for a game engine context, as no universal vocabulary exists yet. It also serves to give an overview of different approaches for the many game development problems, in order to understand why a certain solution has been chosen. The second part will analyze Telesto, a prototype of a game engine architecture. During this part of this document, many links will be formed to the first generic engine introduction, in order to understand the context of the decisions.

As a game engine consists of a wide variety of modules, which are largely independent in their challenges, the evaluation was moved from a general chapter into the specific subchapters. The physics chapter, for example, will include benchmark tests of different collision algorithms. The same happened to potential improvements and planned features. In order to preserve the context, they are described in the specific chapter, rather than a separate listing.

2. Contents

1. Abstract 5
   1.1. About this document 5
3. Games 13
   3.1. Character-based Games 14
   3.2. Omnipresence-based Games 16
   3.3. Hybrid Games 17
4. Architectural Principles 19
   4.1. Game Development 20
      4.1.1. Gameplay 21
      4.1.2. Content 21
      4.1.3. Community 22
      4.1.4. Deployment 22
      4.1.5. Porting 22
      4.1.6. Tooling 22
   4.2. Game Engine Overview 23
      4.2.1. Game Mechanics 24
      4.2.2. Physics 25
      4.2.3. Networking 26
      4.2.4. Artificial Intelligence 26
      4.2.5. Graphics 27
      4.2.6. Interface 27
      4.2.7. Sound 28
   4.3. The in-game universe 28
      4.3.1. Time 28
      4.3.2. Cycles 29
      4.3.3. Constant Cycle Length 29
      4.3.4. Variable Cycle Length 30
      4.3.5. Prediction and Continuous Cycles 31
   4.4. Parallelization 32
   4.5. Causality 32
5. Multiplayer 35
   5.1. Synchrony via Domains 36
   5.2. Synchrony via Lambdas 37

5.3. Synchrony via Reconstruction 37
6. Game Objects 39
6.1. Simple Information Models 39
6.2. Actor Models and Active Objects 40
6.3. Modular Shared Memory Models 42
6.4. Behaviors 43
6.5. Game Object Containers 44
7. Telesto 49
7.1. Motivation 49
7.2. Goals 49
7.2.1. No external dependencies 50
7.2.2. Full .NET Project 51
7.2.3. Maintainability and flexibility 51
7.2.4. Synchrony via reconstruction 52
7.2.5. Prototype 53
7.3. Exclusions and options for the future 53
8. The Telesto Architecture 55
8.1. Visual Basic 55
9. Core 57
9.1. Time and cycles 57
9.1.1. Brush Cycles 59
9.2. Lisp and LScript 60
9.2.1. Avoiding Lisp call-chains 61
9.2.2. Caching 63
9.2.3. Variable Indexing 68
9.3. Resource Manager 69
9.4. Extractors 73
9.5. Input handling 73
9.6. Containers 75
10. Foundation 77
10.1. Entities 77
10.1.1. Multiplayer synchrony-via-reconstruction 80
10.2. Physics 82
10.2.1. Dynamics 82

10.2.2. Collision Geometry 90
10.2.3. Collision Detection 92
10.2.4. Collision Analysis 96
10.2.5. Collision Response 96
10.3. GUI 101
11. Externals 103
12. Thread Map 105
13. Acknowledgments 109
14. Integrity Statement 111
15. Appendix: Benchmark Systems 113
16. List of Figures 114
17. References 115

Part 1

Theory of Game Development

3. Games

Interactive entertainment has become a very powerful economic sector, en par with old-school juggernauts like Hollywood and the music industry (1). Due to its extremely widespread audience, there are countless types or even meta types of games. This ranges from small web-games embedded in advertisements on your favorite webpage, to giant mass phenomena like World of Warcraft (2), which has a serious influence on real-world society (3). It's not up to me to judge whether this development is beneficial or if the "entertainmentisation" of our society is a problem we should think about. No matter how you put it, the gaming industry has become an important factor in our daily lives and there is a lot of development and research involved in creating games. This thesis will try to cover the most important aspects of a certain type of game development - namely the default desktop-PC computer game – as well as feature an exemplary architecture and give some implementation details.

First, however, it's important to clarify what to talk about. There are countless categorization methods for games, aiming at different aspects. Everybody should have read about genres like Strategy Games, First Person Shooters or Arcade, but it is difficult to find a global nomenclature when talking about the full spectrum of games, including merchandise games, console titles and the characteristics of the western and eastern gaming market. For the sake of this document, we will concentrate on commercial PC titles. Many console games released in the past years would fit this classification, too. In fact, quite a good percentage of games are developed for multiple platforms, including both PC and console versions - and in some cases even smartphones. However, the hardware characteristics of a console are different from a commodity PC and therefore require different development approaches. We will discuss this later on, once we take a deeper look at the mechanics of a game engine. Think about memory management, graphics API or multithread capability for example.

Just looking at the variety of PC games leaves us with a large collection of different genres and requirements. Even without any further knowledge, one should easily see that a first person shooter has different requirements compared to a Strategy Game. Just take graphics, for example. The game engine of a First Person Shooter - presenting the game world from the player’s perspective - has a much harder time simulating a realistic environment than a Strategy Game, where you hover above the game world and issue commands. This is a perspective most of us do not encounter in daily life, which leads to a greater acceptance of abstract graphics information while still maintaining the illusion of a realistic world.

Nonetheless, many genres share requirements. For example, the basic role-playing game follows a single hero character or a group of characters and features similar requirements compared to First Person Shooter. Equivalent similarities can be found for other genres, such as Strategy Games and Economic Simulations. The genre classification was introduced from a consumer’s perspective and describes the actual game content or certain gameplay characteristics. In order to compare the engine technology of games, this paper will use different nomenclature, dividing the majority of games into two meta genres: character-based and omnipresence-based games.

In taking this route, we only look at a subset of games, the so called real-time games. What the term "real-time" implies exactly will be discussed in a later paragraph. For now the difference can easily be made visible if turn-based games like chess or Civilization (4) are compared to real-time games

like StarCraft (5). Each of these three is a Strategy Game, but chess and Civilization (4) do not simulate a "constantly running universe". Although most of the technologies discussed in this document can be used for turn-based or other non-real-time games, there might be better solutions, as many problems of simulating a real-time-universe do not arise if the continuous constraint is not required by the gameplay. But I'm getting ahead of myself. Today, the gaming market is strongly dominated by real-time games and they are definitely the most difficult task to look at.

Now that the classification "real-time" is fixed, we still have to think about the little world, "game". A more scientific description for the games we are looking at would probably be a "soft real-time interactive agent-based simulation". Let's break this phrase down to understand each part. We already know what real-time is in the context of a computer game. In contrast to hard real-time, soft real-time means that the simulation accepts delays. There are certain limits in every game, for example, 60 frames should be rendered per second. If the system is slow, this might drop to a lower frame rate. If this happens, the simulation does not die. An example for a hard real-time system could be an artificial heart pacemaker, which does not tolerate delays at all.

A game is also an interactive simulation. It is not reality. It's a simulation of reality, using abstraction and simplification to achieve a mathematical model which may be processed by a computer. This is also true if the "image" for the simulation were a fictional universe and not reality. It still leads to an abstracted and simplified model. As a game involves human interaction, it also has to be interactive.

As games try to represent entities of some kind, they are agent-based, with the agent modeling the universe's entities. There are generally multiple agents per entity, but we will look into entities - called "Game Objects" in the context of game development - later on.

3.1. Character-based Games

Many games follow a player's character of some kind, be it a sword-wielding hero, a racing car or a spaceship with its crew, which the player controls more or less directly. The most important part is that a player interacts with the world by controlling his or her character. Keeping this pattern in mind, we can identify certain key characteristics of a character-based game engine.

A character-based game engine can utilize what is often called "spheres of influence". A sphere of influence is the maximum region a player's character can cause events in. As a character is a spatial entity which has a physical representation in the in-game universe, it is bound to the in-game causality paradigms. The sphere of influence of a player's character in a First Person Shooter could, for instance, be its maximum viewable sphere, defined by the maximum viewable distance, as a player can't shoot things he or she cannot see. Of course, if the bullets fly long enough, it's possible, but most games delete bullets once they exit the maximum viewable distance or after a certain period of time, given that they are simulated as projectiles and not just basic ray casts. The ability to expect a sphere of influence, no matter how exactly it will be calculated or handled, allows a lot of architectural tuning to increase performance in the field of simulated causality and multiplayer synchrony. Spheres of influence are not always visible directly in the code, but they are the key concept that allow causality-tuning for character-based games. Causality will be covered more in depth in later chapters.

Another important point about character-based games is the addiction to graphics - or better said the simulation of a believable environment. We encounter daily life as a character-based game, too.

Gameplay might not be the best, but the graphics and physics are pretty neat. Therefore our brain is well trained to perceive and analyze an environment from our own perspective. We are also trained to find unexpected or strange (in the sense of uncommon) things within a first-person environment, since every strange noise or unusual leaf movement could have been a saber-toothed tiger on the prowl. Therefore we are very capable of finding clipping errors, UV-seam mismatches or stuttering animations. If the goal of an engine is to simulate a photorealistic environment, this becomes a severe issue. Many modern games break with photorealism, by utilizing comic-style graphics for example. This bypasses our brain's link to real-life and stops it from searching for saber-toothed cats - and from finding rendering flaws in the process. This is another good example for the ongoing content versus technology problem that one can find in almost every aspect of game development.

The following list contains a few typical character-based genres with examples. These games were also useful for my studies while working on the engine. These games were chosen because they are either popular representatives of their genres, their source code or other useful information is published, or because I simply enjoy playing them. There is no reason why game-development shouldn't be fun. Some notes of why I chose the respective title are given next to name. There are a lot of other potential games out there and this list could easily be doubled or tippled in size, yet it is best to focus on a limited collection of examples. Starting to work on game engines could begin with playing a few of these games and taking a closer look at what's actually happening behind the scenes. One can notice a lot about the actual engine without even reading a single line of source code.

First Person Shooters	Half-Life 2 (6) *Source Engine*, Game-Mechanics source code published, powerful SDK Crysis (7) *CryEngine 2*, background information, superb renderer, editor Counter-Strike: Source (8) *Source Engine*, contacts, level design, balancing Killing Floor (9) *Unreal Engine*, RPG-Shooter mix, hell of a lot of fun to play
Roll Play Games, Adventures	Neverwinter Nights 2 (10) Unique engine, complex game mechanics, SDK
Racing	FlatOut 2 (11) Unknown engine, good physics and damage model, multi-platform
Massive Multiplayer Online Games	World of Warcraft (2) *"Warcraft Engine"*, MMOG server model, Content vs. Technology Aion (12) *CryEngine*, large populations management, using a shooter engine for an MMOG Guild Wars (13) Unknown engine, "faking" a massive universe, Content vs. Technology

Hack'n'Slay, Action Adventures	Sacred 2: Fallen Angel (14) Unnamed engine, integration of PhysX (15), Community Management, Interviews
Jump'n'Run	New Super Mario Bros. Wii (16) Unnamed engine, console title, one-console-multiplayer, Wii-Remote experiments
Flight Simulators	IL-2 Sturmovik (17) Unique Engine, AI, atmospheric simulation
Sports	(I did not look at this genre, but most games would fit)

3.2. Omnipresence-based Games

As the name already indicates, an omnipresence-based game does not directly tie a player to a certain game entity. Instead, the player exists as an abstract not-spatial concept and interacts with the game world by influencing the in-Game Objects via certain tools. A popular example for this approach would be Strategy Games, where the player moves across the map as some kind of god-like entity and issues commands to his or her army. Interaction with the in-game universe takes place as the armies are on the move and execute their given orders.

Due to its non-spatial characteristic, an omnipresence-oriented game engine can't draw spheres of influence around a player. In most cases, a player could interact with any of the present objects in the complete in-game universe. However, the channels of interaction are usually strictly defined. Therefore we can draw layers of influence, as only certain actions can be influenced by the player. Utilizing the above Strategy Game example, a player can give his or her tanks a move order or an attack order. However the player can't precisely control the orientation of these units or where the guns will aim. Layers of influence can be seen as communication channels. The player invokes a certain action by sending a message on a given channel, but there are a lot of actions which happen as a result of received massages and are not directly player-influenced. This is unlike a character-based game, where a Game Object can usually be shot at any time. We will discuss causality and synchrony later as we speak about the core features of game engines, but keep in mind that the "omni" part of omnipresence-based games will stress the Game Object systems quite a bit, as all entities in the game would have to expect to be influenced by the player at any given time.

Strategy games look ugly. That's a common prejudice and there are in fact good examples to support this point. Yet there are also good examples of modern Strategy Games (or similar omnipresence-based genres) which simulate a believable, realistic environment. That's due to the fact that we are not transcendent yet and are not used to an omnipresent view of the world, such as floating above the battlefield. Therefore our brain automatically assumes some kind of abstraction and accepts less detailed, conceptual graphical information, without trying to find issues or hungry cats. Compare the issues of simulating a forest for example. In a First Person Shooter, you will relatively quickly notice that it only consists of, let's say, ten different tree models, while a Strategy Game can produce good vegetation results with far less diverse models. This of course also has to do with the perspective. In an omnipresence game you are zoomed out and you will always see many more trees at the same

time, which reduces the required detail on a single tree. Typically, omnipresence-based games have fewer graphical and environmental requirements compared to character-based games.

Once again, let me close this paragraph with a list of games I have chosen as references. Just as I did for the character-based games, I will only state a few key notes on why I chose them and what my focus was while analyzing them.

Real-Time Strategy Games	Star Trek Armada II: Fleet Operations (18) My own game, based on Star Trek Armada II (19) StarCraft II: Wings of Liberty (20) *"Warcraft Engine"*, MMOG-like features, integration into social gaming Supreme Commander (21) Unnamed Engine, scripting language, massive physics simulation Homeworld (22) and Homeworld 2 (23) *"Homeworld Engine"*, incredible clean source code, Game Object model Sins of a Solar Empire (24) *Iron Engine*, giant object counts, fractal generation, technologies to handle both tiny starships and giant planets
Economy Simulations, Economy Games	Anno 1404 (25) Unnamed Engine, renderer (post-production effects), Game Object counts, multithreading, interviews Die Siedler II - Die nächste Generation (26) Unknown Engine, reviving a ten-year-old game with new technology
Tower Defense	Defense Grid: The Awakening (27) *Gamebryo Engine*, no multiplayer?!, release mechanisms, interviews

3.3. Hybrid Games

There really seems to be no rule without an exception. This meta-genre classification is no difference. The borders between characters and omnipresence are not always as clear as they are when comparing a real-time Strategy Game to a First Person Shooter. Consider a tactical shooter for instance, where a player commands an elite unit from an omnipresent perspective, but is only able to influence the environment through his or her given handful of heroes, while still able to scroll around the whole map. Wouldn't it be possible to draw a sphere of influence? However, it's necessary to process Game Objects outside the sphere of influence, as the player is still able to see them. Such a game could of course just use an omnipresent architecture, but a hybrid architecture which takes these unique gameplay characteristics into account might better fit this scenario and could free up processing time for extra effects.

Another example could be a role-playing game, where the player follows a group of heroes. If the heroes split up in a dungeon, the player might be allowed to switch between them. This could lead to very vast spheres of influence, pressing right up against the border of an omnipresent perception of the game universe. The gameplay could solve this issue by dividing the different paths the characters chose into different chapters, where the characters are played one by one and only

influence each other via certain channels until reunited again. Nonetheless, an architecture that supports rapidly changing spheres of influence could lead to some very interesting options.

The hybrid nature of a game becomes very obvious if the mere genre description already classifies it as a hybrid. My favorite, although quite aged example, is Battlezone II (28), a genre mix between a First Person Shooter and a real-time Strategy Game. Players command a character from a soldier's perspective and may use vehicles like tanks and mechs. Yet, they also build up a base, gather resources, produce new vehicles, and command squads. To do so, a commander is assigned, who may enter a bunker, upon which the game switches into an omnipresent perspective. The commander may now issue construction orders and assign units to squads, which are lead into battle by another player from a first-person perspective. It's very interesting to see these options combined. For example, the rendered details are reduced once the bunker is entered, the GUI layout changes completely, and even the way Game Objects interact is altered. Similarly, First Person Shooters have a crosshair instead of a mouse cursor, and mouse movement is not mapped on the GUI-cursor movement, but is a direct Game Object state alteration (such as having a player face in a new direction). With the "first person" mode, the channels to interact with an AI-controlled squad unit are very limited - basically just a rough waypoint system. Omnipresence mode offers far more interaction. Due to the age of Battlezone II (28) it runs well on all modern machines. Conversely, back when it was released, the omnipresence mode was quite slow. Most interestingly, there are no modern games which utilize this gameplay pattern. A slight hybrid touch can be found in newer shooters too: for example the commander mode in Battlefield 2 (29). However, that's far away from the complexity of a real "50:50" hybrid.

Real-Time Strategy Games and First Person Shooter Hybrid	Battlezone II (28) *Battlezone II Engine*, rapidly switching requirements, complex Game Object model, both Star Trek Armada II (19) and Battlezone II (28) are based on the same predecessor engine: Battlezone (30)

4. Architectural Principles

We will now start thinking about the actual architecture of game engines. If you have already read papers or a book about this topic, you might already have encountered contradictory vocabulary or even different definitions for what a game engine actually is. That's fine. Engine development is not a science, and there is no global nomenclature. Every developer has his or her own private view of the topic and that's actually an important point, as it allows for new views, methods and ideas to pop up. There is a giant collection of definitions about what a Game Object actually is, for example. Everybody has his or her own reasons and special view on the topic, originating from the usage case of the engine. A character-based developer will have a different view on an in-game universe than an omnipresence-based developer would have. However, if we want to discuss these topics, we have to agree on certain terms. This theory part will supply the required definitions, which also match the view I had when developing the Telesto engine architecture shown later.

So, what actually is a game engine? From a historical perspective, the first games consisted purely of hardware. The first software games like Pac-Man (31), Tetris (32) or Space Invaders (33) were directly implemented and their code could not be modified to play a different game, even though Tetris (32) or Space Invaders (33) feature quite similar gameplay. Soon, game developers began to use configuration files to describe parts of a game's behavior, like Civilization (4). Its Successor, Civilization II (34), was actually the first game I started to mod. It used ini-files to configure buildings and units to produce and I made a small space-based mod for it. This type of development allowed the creation of very similar games based on the same game engine. Still, the term "game engine" arose years later, in the 1990s. The first game deploying a real engine, as we use the term today, was probably Doom (35). It gave birth to the first incredibly popular engine series, the Quake (36) Engine, with more than 50 games released until the current day, covering several genres and gameplay characteristics, although the vast majority are First Person Shooters. Even the Source Engine, the engine of the popular story First Person Shooter, Half-Life 2 (6), is probably based on the Quake engine, which was used for its predecessor Half-Life (37). Another popular engine family is the Unreal Engine, which was first used in its name-giving game Unreal (38). While the engine market nowadays seems to be dominated by First Person Shooters, that's not actually true. There are a lot of engines used for Strategy Games, like the unnamed engine used for Warcraft 3 (39). There are strong hints that the engine of World of Warcraft (2) and StarCraft II (20) are also based on the Warcraft engine, but this has not yet been confirmed officially by Blizzard.

Today, there is even a market for selling engines without games. Companies like Crytek develop a game like Crysis (7) as marketing for their actual game engine, where they get the majority of their profit. The large capacities of the Crysis Engine are demonstrated by Aion (12), an MMOG, while the CryEngine was originally developed with First Person Shooters in mind.

For the context of this paper, a generic definition for a game engine has to be found, to cover at least the current perspective on this changing field of development. For me, a game engine is another abstraction layer, which offers an application programming interface (API) to manage an in-game universe. A game engine encapsulates the actual collections, shared memory blocks, behaviors or whatever approach one might think of and offers it in the form of Game Objects, universe constraints or similar structures to the developer. A game engine allows the creation of different games (probably of the same genre or metatype) on the same technological foundation. This invokes

an important question: are there modern games without an engine? In my opinion: yes, indeed. If you start developing your first game, probably without in-depth knowledge of your chosen programming language and environment or even without too much coding experience, you will start with an idea of your game in mind. You will begin at some point, probably with the graphics, and begin implementing your features. In the end, you might have mapped your idea onto your harddrive in the form of a playable game. Yet, you will have implemented the features just to serve their purpose in your idea. There won't be much abstraction and no real architecture. That's not bad in the first place - just look at the incredibly popular old-school examples like Space Invaders (33). However, games developed using this pattern typically cause a lot of trouble if logical game-mechanics bugs arise or the game concept is extended. Sometimes it's not even possible to do so.

To define the game engine as an abstraction layer offers another question: what does the game engine include? That's a very tricky inquiry to answer. Basically, the answer would be that it varies from engine to engine. All engines offer an abstraction for the basic entities, often called Game Objects, which map the logic of a game. They are usually closely linked to the game mechanics. However, what about all the other aspects of a game engine like the renderer, sound, physics, AI and much, much more? In some architectures, these are integrated into the core engine. Others just offer APIs to communicate and declare stuff like the renderer as an external module. For a clean definition, let us agree to the following in the context of this paper:

A game engine is an abstraction layer which offers an API to create, interact and destroy a game universe. To do so, the game engine offers structures or APIs to control entities in the game universe.

This definition leaves the question open as to whether the presentation of the game universe is a job of the engine itself, or external stuff, such as a separate graphics engine. In this paper, we will also take a look at topics which could be encapsulated and be threatened as externals, like physics. In the game industry, the choice as to what to include in development is usually a budget question. Renderers are almost always developed as part of the engine, as they are very important key presentation points to sell the engine later, while physics are often bought as third party packages and only integrated into the game engine via an API or by synchronizing with a separate physics universe managed by the physics engine. Nevertheless, a game engine will have to define the principles to work with these external modules. Therefore, most game engines deploy a central processing core, which establishes the runtime behavior of the game universe. It is often responsible to manage time and causality as well as maintaining Game Objects or similar modules.

4.1. Game Development

Before we continue, let's take a brief look at the things that, no matter how you precisely define the term game engine, are not considered part of a game engine, but are required to actually bring a game idea from the sketches to the shelves. It is important to understand the links and requirements a game engine has to provide in order to be deployed in a large-scale project like a commercial game.

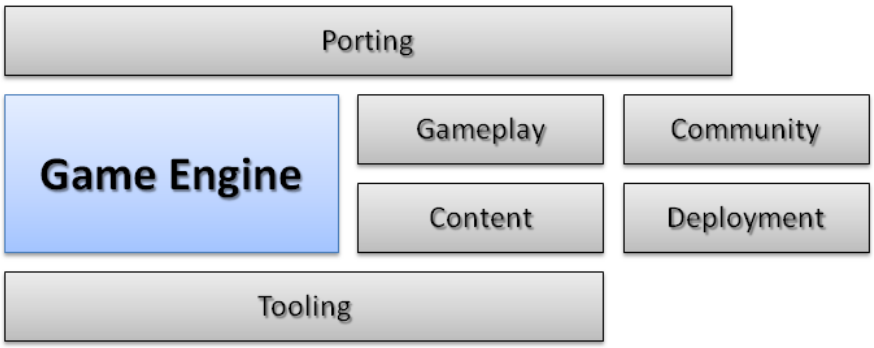


Figure 1: Game Overview

4.1.1. Gameplay

The rule set a game follows and the mathematical models behind game entities are often called game mechanics or gameplay. For the purpose of this paper, I will call the scripts that actually enforce the rules of a game the game mechanics, while the idea behind the game - the concept that a game designer originally had in mind - is called gameplay. The game mechanics are obviously linked to the game engine, as they are part of the in-game universe. On the other hand, depending on if you count scripts as content, at least the API enabling the game mechanics scripts are part of this universe.

4.1.2. Content

There is a large difference between an application and a game. An application usually does not contain its own content as customers create the content with the tool themselves, while game development includes creating both the content and the device to work with it. Content usually includes meshes, sounds, levels, interface graphics, but there could also be game specific content domains. A good example are voxel-models required for some of the new voxel engine experiments recently popping up, like the id Tech 6 engine (40). These types of materials are easily identified as content since the game engine uses them to make its current in-game universe state perceptible to the user. However, there are other types of content where this definition will not work out, such as scripts of a scripting language used to describe the behavior of Game Objects. These scripts are closely linked to the game engine itself, as they define or influence the way entities in the world interact with each other. It's a design decision as to whether or not one counts them as content. I will not classify scripts separately in this paper, but instead count them as part of the entity they describe or where they are used. Game Object scripts are part of the game engine, while scripts defining material properties for a 3D renderer are content.

At the end of the day, content is one of the most important aspects in game development. Content, particularly the graphics and audio content, is the first thing a player comes into contact with when playing a game. Bad content can literally break a game before the player even starts discovering the potentially superb gameplay. On the other hand, good content does not make a good game. There are many examples of very popular games, like Warcraft III (39), which utilize older graphics content (even at the time of their release), in order to reduce the hardware requirements and in doing so have increased the potential audience, especially in the casual gamer sector. However, there are far more examples which use state of the art content, but still fall into silence after a short hype period.

While content quality is very important for the first impact of a game, good gameplay is responsible for generating long-term gaming experience.

4.1.3. Community

With the growing popularity of social networks and massive multiplayer online modes even in typical singleplayer genres, building up a community and the resulting community management are key sales points in modern game development. At first glance, this sector might look completely independent from the technical aspects of a game engine. If a game engine does not produce competitive results - for example, due to an old renderer - it might of course be difficult to establish a community. Yet the community also has technical requirements an engine might have to take care of. Most Massive Multiplayer Online Games (MMOGs) offer in-game customer service, called gamemasters, to aid and assist players within the in-game world. These features have to be taken into account when designing this type of engine architecture. If they are not, they would lead to additional security requirements in order to establish in-game administration features, such as those required for gamemaster services.

4.1.4. Deployment

To actually sell your game, you will have to bring it onto the shelves (real or virtual ones). This involves a lot of marketing, publishing and release processes, which I don't want to cover here. One could easily write a number of theses about this sector. I have simply summarized it as deployment. Similar to the communities sector, there are a lot of links reaching from the game engine to deployment. They are quite obvious: a good game is more easily deployed than a game with technological problems (marketing can blur a lot here). In the context of this document it's more interesting if there are also links from the deployment that might reach to the game engine, of which there often are. Consider developing an MMOG again, where "selling" the game means selling play-time. An MMOG usually consists of two important parts. A client running on the player's machine, and a server, running on a larger server cluster. The server simulates the in-game universe, while the client is basically just a screen to display the current universe state. While the game is running, a lot of information is accumulated on the server. The way the server-engine deals with this information can have a large influence on deployment, such as the maximum capacity of a server, potential weekly downtimes or external tools like web services to view character information on your smart phones. If you Google the most popular MMOGs, you will find a lot of these features to be very important arguments to gather a community.

4.1.5. Porting

Developing an engine for multiple platforms obviously creates a lot of requirements. Consoles and desktop PCs usually have very different hardware. This is especially visible in memory management and graphics hardware. This problem gets even more severe if you also take smart phones or web-based games into account. If it is decided to develop a game for multiple platforms, there will also be an impact on gameplay, as different devices require different controls, like a multi-touch tablet for example. Portability might also influence community management, due to a changing audience.

4.1.6. Tooling

Often forgotten, but still very important: developing the tools to work with the game engine. Time-To-Market has become a valuable development parameter and allows a development team to react on trends in their community, which is very important for entertainment products. Spending more

development time on tools and content-creation-processes is also necessary if a goal is to sell the engine for 3rd party development projects. Modern tools indeed require being taken into account during engine development. For example WYSIWYG editors require a renderer that is able to work on both static (level) geometry and dynamic (level editor) geometry.

4.2. Game Engine Overview

Game development is not science, it is handcrafting. There are no strict rules or guidelines one can discover that will always work out. There is not even a generic approach. For this very reason, there are countless game engine implementations, countless approaches and countless views on hierarchy. It would be far beyond the scope of this paper to talk about them all, especially because new ones pop up each year. Nevertheless, we have to take a generic look at game engine architecture first, in order to go further into the details of an implementation and to establish a common vocabulary. The following chapter will show and explain an exemplary diagram of a game engine by dividing its modules into different functionality sectors. The modules listed here can be found in most game engines. Sometimes a few of them are fused together, or a new module replaces one or more existing ones, but the basic functionality is always present.

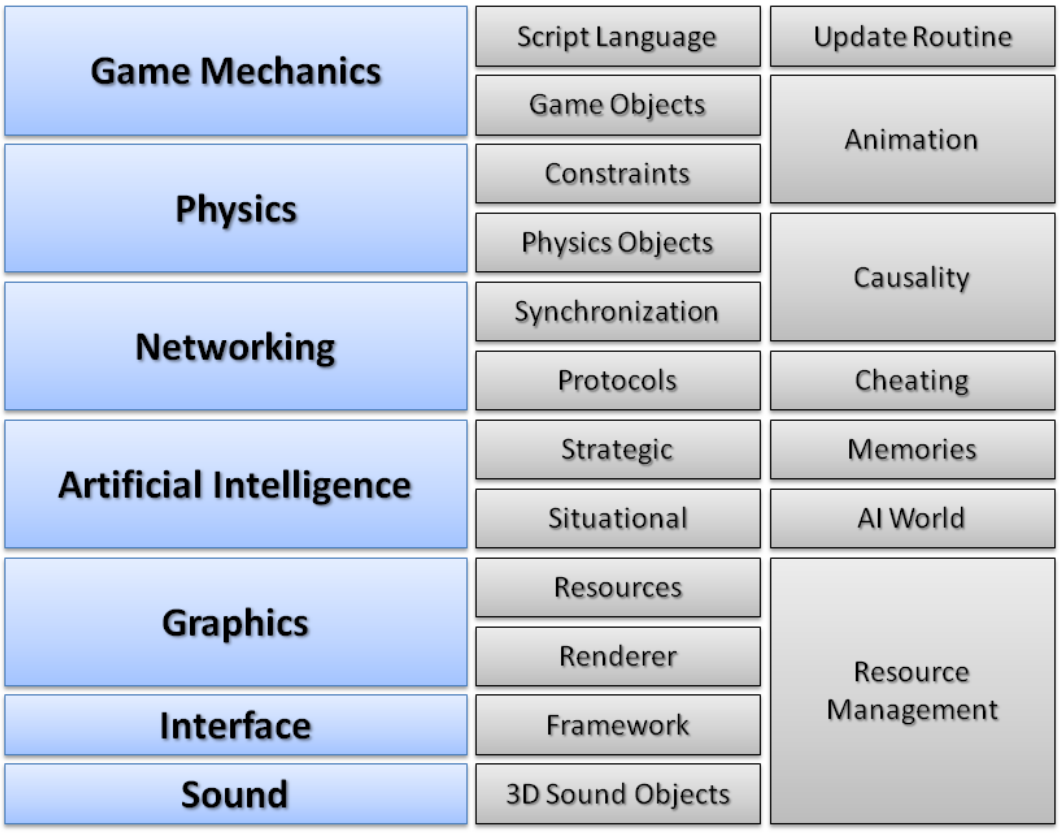


Figure 2: Game Engine Overview

The left side of the diagram shows the seven core modules a game engine has to take care of, while the right side contains the most important aspects or features of the respective module. We will now take a brief look at each module and what it incorporates, as they are closely mapped onto the Telesto Architecture that we will study more closely in the second part of this document.

4.2.1. Game Mechanics

The most important module for a game is of course the module that maps the game logic and ideas defined in the Gameplay documents onto structures which may be processed. There are games that may consist almost only of the game mechanics module. The presentation, graphics, sound and interface, of a chess simulation, for example, will probably be very thin, and AI, networking and physics could be skipped entirely. Such a lightweight chess simulation only consists of game mechanic descriptions of the different figures, a game round, and allowed player moves, all of which build up the chess universe. There are no real games which could work without a game mechanics module. Physics games come into mind, where the player has to solve puzzles using gravity or collisions. Nonetheless, even this approach requires the concept of a game round to determine victory conditions, unless the game is a mere physics sandbox, where the classification, "game", might not really be appropriate.

The vast majority of games nowadays represent the game mechanics encapsulated into Game Objects. They might not be called that way, but there is always a structure that describes the behavior of a game entity in respect to its game mechanic role. Good examples are player entities which may carry and fire weapons. The process of picking up an item and putting it into an inventory is a game mechanics process, as well as firing the actual weapon - at least as long as the physics simulation is not as precise enough to simulate the complete firing process. Current physics engines are not capable of simulating this complexity in real time, and it is probably safe to say that this won't change within the next few years.

The Game Objects are often the most changed aspects of a game, especially during the post-release maintenance period where bugs and balancing issues are solved. Balancing changes in particular always have a large impact on the Game Objects or their supported features. The origin of this relation lies between the ideas behind the different modules. The requirements of a physics engine can relatively precisely be described during the development phases without much need of alteration in later development stages, because physics engine have a lot of capability for generalization. They implement a certain physics model, for example, rigid bodies with Newtonian collisions, without having to know much about the precise usage scenarios and the game design later on. The game mechanics, on the other hand, are closely linked to the view of designers on the gameplay. It is exactly these ideas which change during development as new solutions for gameplay problems come into mind or as balancing issues have to be considered. Due to the high probability for changes, the game mechanics are often described using a scripting language, as scripts are much faster to develop and mechanisms can be deployed to make the actual runtime core stable even if changing scripts, by narrowing down the capacity of what the scripting language can access. Scripting is also a benefit for tooling. A level editor is basically a tool to create Game Objects. A scripting language allows complex actions to be designed by a level designer, like hacking into a computer console or dynamic changes of the environment, without needing to know anything about the actual Game Objects behind it. In reality, this does not hold true, as level designers require performance relevant information and guidelines on how to use the scripts. However, these performance operations could also be integrated in the scripting language API. A scripting language also makes it possible to build a modding community.

Last but not least, the game mechanics include the routine which keeps the Game Objects in temporal coherence and transforms the in-game universe as time goes by. The update routine is covered in the in-game universe chapters.

4.2.2. Physics

Simulating believable real-time physics in games has become very popular in the past few years, at least for the purpose of adding eye candy, like debris flying away after an explosion. Nonetheless, many titles include third party physics solutions, as developers avoid programming the complex physics engines themselves. Popular physics engines are PhysX (15), Havoc Physics (41) or the open source ODE (42). With access to general purpose graphics cards growing for commodity PCs, we will probably see new physics effects and improved precision in the next one or two generations of games.

The field of physics is vast. When game developers talk about physics or physics engines, they usually mean dynamics simulations with a collision detection and response system. In most games this will be a rigid body dynamics simulation, but we already have titles around with soft bodies, cloth, and liquids. To clarify: dynamics describes the forces (and their resulting torques), which cause movement and action of objects (the kinematics of entities), while a collision detection and response system handles the interaction of objects if they collide. Unlike our real universe, where objects are delivered with their inherent properties (like being solid), a game developer, being a deity in his or her own in-game universe, has to take care of all these things and tell the objects precisely not to intersect. I will follow this game development tradition and refer to a dynamics simulation with a collision system as physics in the context of engine development. Even if no dynamics simulation is used in a game (consider an old Jump'n'Run for example), there will still be a collision detection and response system in every game, otherwise our plumber would fall through the floor. Yet a game engine is only considered to have "physics" if a dynamics simulation is present.

All physics engines have to deal with at least two constructs. These are Physics Objects and constraints. Much like Game Objects, Physics Objects are entities representing the physical aspects of the entities in the in-game universe. Depending on the chosen physics engine, this might either be just a query system to access the state of entities in a separately running physics simulation or a handcrafted entity model within the game engine itself, where interfaces allow the running physics simulation to adjust values such as object transformations. The other important element the physics has to deal with are constraints, such as atmospheric dampening or relations between objects. A good example is the turret of a tank and the tank body. While both are separate physics entities, they are linked to each other. Physics events happening to the tank body also have an influence on the tank turret. There is a lot more to say about simulating physics, but I will keep this paragraph short. There will be more information on physics implementations as we take a look at an example in part two of this document.

We have just started taking a look at game engines and yet we have already found an important point of conflict. Both the Physics Objects and the Game Objects describe different aspects of the same in-game entity. Shouldn't there be actions where both of these aspects might conflict? This is indeed a major problem of game engine architectures. The Game Engine Overview shows animations as an example for this problem. A player might control his or her character by pressing movement keys. This is induced by the game mechanics, yet it has an impact on the physics, as a

player is a physical entity. If a player tries to move through a wall, there is a conflict. The game mechanics wants to move the entity further, while the physics engine calls for a collision. Policies and algorithms have to be established to solve these issues. In this example, the policy might just be "physics first", to ensure a player cannot pass through solid objects. Other animation examples, like a mage moving its arms in a dramatic gesture to cast a fireball might just ignore physics altogether and have the arms pass through walls if the mage stands next to a building, as these animations are not that important for the overall physics simulation. Other examples might need the physics and game mechanics to negotiate. For example, this might be the case if an object is to pass through a tenacious liquid, where game mechanic inputs (like player movement) influence the physics simulation, but the physics engine still has a word in the final simulation result.

4.2.3. Networking

If multiplayer - except multiplayer modes on a single console - are required for the gameplay, networking will have to be part of the engine. While it might sound simple at first glance, it is one of the most defining questions when designing game engine architecture. We will take a closer look at the synchrony problems in the following chapters, including mapping causality in order to solve the multiverse problem. In other words, keeping several game instances synchronized. I have dedicated a whole chapter to multiplayer later on in the thesis.

The pure low-level aspects of networking - transferring data between clients - are identical to problems encountered in server clusters or office applications. There are many good books about these fields and I won't cover this aspect of game development in this document.

4.2.4. Artificial Intelligence

Creating a good AI for games is a very interesting and challenging objective. Nevertheless, I had to skip this sector, as you could easily write multiple documents just about AI design. I will still try to summarize the basics.

There are usually three different classifications for AIs in games. Assistance AIs are usually mere scripts or routines deployed to establish functionality. If a player in a real-time Strategy Game issues an order, the commanded units have to move to the target location, avoiding obstacles along the way or engaging opponents in a target area.

Situational AIs are often just parameterized scripts which evaluate their results based on short term memories. For example, this is used to implement autonomy functions or pure tactical opponents. Consider a real-time Strategy Game again. If two tanks engage each other due to a player's command, a situational AI script could look for nearby buildings in which to automatically take cover. Another example are bots in First Person Shooters. Most First Person Shooters require only a few long-term memories, such as defining a strategy. They are usually just tactical decisions made with precompiled script data. That's just the way most modern shooters implement their AIs. Each map includes information for the AI, like movement paths, points of interest or power-ups. While playing the game, an algorithm makes the bot move along these paths or from region to region until it engages an opponent or difficult situations. Situational scripts will now attempt to solve the issue by firing a rocket launcher or jumping over a wall. After the situation is resolved, the memories are discarded or boiled down to a small set of attributes to alter the behavior in the next engagement.

Strategic AIs are very rare. That's probably due to the fact that they are very hard to implement. There is in fact not a single game with a real-time strategic AI. Most games use precompiled data, like the bot-maps in the last example, and situational scripts to solve unforeseen situations. This works well for most genres. Real-time Strategy Games would - the name suggests it - be the perfect test bed for strategic AIs. A strategic AI accumulates memories throughout the whole game and has little precompiled information. In the above First Person Shooter example, a strategic AI would probably be supplied with information on what different items and weapons do, but no information on the map itself. While playing, the AI discovers the map and learns about the human players present. It could adapt to certain patterns which are used by players - like camping with a sniper rifle - and react to them. While this might sound a bit like science fiction, I'm sure we will see the first strategic AIs in games relatively soon.

4.2.5. Graphics

Graphics are a well explained and important part of game engine architecture. Many books about game development are actually books about developing a graphics renderer. This document will not cover the actual renderer development, as it is both very platform and game genre dependant. However, we will take a look on the interfaces required to connect a renderer to the game world and the methods used to extract the current state of Game Objects to present them.

The renderer also requires a lot of resources: textures, meshes and materials. That's why the renderer is usually very closely linked to the resource manager. A resource manager is required to load and unload large resources from the hard drive. In most cases, a resource manager creates a virtual file system which holds links to the files and manages their loading and unloading. In order to hide the actual file access from the rest of the engine, asynchronous loading is used. This system works by having files either flagged as loading with a caller deciding what to do on its own, or a default resource is returned while the requested file is not ready.

4.2.6. Interface

Game development generally divides the user interaction into two fields: the GUI and the HID handling. A human interface device (HID) is everything connected to the PC or console that allows a player to interact with the game universe. For PC games, this is the keyboard and the mouse, as well as other controllers, such as game pads, joysticks or exotic 3d controllers like the Space Navigator (43) from 3Dconnexion (44). Consoles usually bring their very unique controllers. For instance, a classic gamepad or 3D controllers like the Wii-Remote (45). A more recent development is multi-touch-multi-user HIDs. A game engine has to receive events from the connected HIDs and translate them into an internal format for further processing. For example, this could be by sending out button-pressed-messages to an in-game entity which may invoke state changes of other entities, depending on the pressed button. An exemplary interface, including HID handling and a basic GUI framework will be presented in the second part of this document.

The graphical user interface (GUI) is the collection of all buttons, bars and text which float "before" the actual game world, in order to offer interaction channels to a player. The complexity of the GUI varies greatly from genre to genre. In a First Person Shooter, the GUI is just a device to present some information, like ammunition and health, while in Strategy Games or MMOGs, the GUI is the main point of interaction between the player and the game world. A simple GUI can be accomplished via a bit of rendered information and a Game Object or two managing it, while a complex one should be

implemented via a GUI framework to be customizable. This is also a benefit, as the GUI of an MMOG, for example, often changes during the game - something that is better implemented in an API or framework rather than hardcoded constraints. A customizable GUI might also allow players to develop their own interface additions, which could bring up a new option for community management. For instance, World of Warcraft (2) has a large modding community, dedicated to creating new GUIs for the game.

4.2.7. Sound

Sound engines are often bought as third party software, similar to physics engines. For example, a very popular one is FMOD (46), used in Fleet Operations (18), a project I am helping to develop, or commercial titles like StarCraft II (20). We will not analyze the details of creating believable sound environments, managing sound channels and using sound hardware. Instead, we will concentrate on connecting the sound engine to the architecture. The basic patterns are very similar to the graphics interfaces and we will discover large similarities in resource management, too. We will identify a generic approach to connect external modules, called extractors, which might also be used for other data mining procedures in an in-game universe, such as gathering information for a web service for an MMOG. However, let's first talk about this in-game universe that has popped up so many times now.

4.3. The in-game universe

There is a great difference between developing a business application and a game engine. While a business application usually serves several uses and might perform very complex programs (even more complex than what we would encounter in any game) it still has a very different character: a business application is a tool. It exists to be called at certain points in time, perform its job and then close again. Its current state is usually closely tied to an input data set (be it saved data or direct arguments passed along with a job by the user).

A game, however, simulates a universe. Much like our real universe, it is an ongoing process, where actions take place due to actions performed by a player, or just because the rules of the universe define them to happen. The current state of the in-game universe is constantly changing and, besides the initial start state of the universe, the rules are the most important influence on the current state. That's a fundamental difference, as it requires a continuous process.

Designing a game universe is basically a bit like playing god in your own domain. Before you can start inventing any type of game entity, you will have to kick off your own universe. The fundamental currency for actions is time. Let's take a look at time and how to implement it as our first game universe feature.

4.3.1. Time

In our daily life, time is quite a boring thing. It always moves in the same direction - from the past to the future - it always moves at the same speed and it always has a defined, unique point called present, between the past and the future. Besides these basic principles, we also have to take care of one principle which we experience in our daily life, but don't usually think of: time is consistent throughout the universe. All entities which are linked to a certain point in time should have the internal state of the defined point in time. In other words: if it is 9 o'clock on top of the desk, it

should also be 9 o'clock under the desk, or on a different continent (yep, our measurement methods for time change as we fly around the globe, but that's just the unit, not the magnitude).

Having time moving in one direction and introducing a defined point called "now" is rather trivial. The two tricky things we have to care about are the continuity and the coherence. Let's start with continuity. Is it possible to write a computer program which simulates real, continuous time for a simulated universe? Sadly, no: at least not with the same speed. That's quite easy to understand as a computer program is just an abstraction for operations on your hardware which are physical action/reaction chains. In order to produce a physical action/reaction pair, we require a certain amount of time. If we want to simulate the time it takes to perform an action/reaction pair on a simulated CPU we will have to do more real life action/reaction pairs to achieve the necessary information. Therefore a continuously simulated in-game time would become incredibly slow. So, what's the solution? Like all real-time applications, game engines skip the continuous constraint. By doing so, we define our own derivative of time that we will use for the in-game universe. That's important to notice, as, at this very moment, we agreed to start moving away from the real universe. God's PC seems to be quite a lot faster than ours!

Sliced in-game time works as a chain of points of time. One of these points in time is called "now". To get to the next point in time, we perform a transformation on the complete in-game universe (including all entities, spatial or abstract), which transforms the state of the universe into a new state, which is a certain amount of in-game time in the in-game future. Once the transformation is completed, we arrived at the next point of time, which could now be called "now" and is a certain distance (which is larger than zero) away from the last "now". However, performing the in-game universe transformation took real time. There are two different patterns to deal with this situation.

4.3.2. Cycles

The transformation from one point of in-game time into the next point of in-game time is often called a cycle, and I will follow this for the purpose of this paper. We will encounter cycles at many occasions. Their management becomes especially important when talking about multi-threading. During engine design, performance requirements are usually measured in time/cycle, such as "we have 3 milliseconds per cycle to spend on physics".

4.3.3. Constant Cycle Length

One way to manage cycles is by defining the in-game-time elapsed between two cycles as a constant value. So we now have three situations to look for. Either we spent less real-time to compute the next "now" than our in-game time elapsed between the two points of time, we could have spent the exactly same real-time, or we took longer to simulate the next in-game-time.

Real-time is less than in-game-time. In this case, we replaced the old "now" with the new "now" too early. In other words, our in-game time became too fast. As the speed of the in-game time will always vary a bit between two timesteps, that's nothing too serious, as long as the average speed of time over a longer time span, which is noticeable for a human player, remains constant (a second for example). All we have to do in this scenario is to wait until the real-time catches up with the in-game time. Otherwise we could produce more and more points of times too quickly and end up with a noticeable speed change.

Real-time equals in-game time. That's our perfect scenario. We took exactly as much real time to transform the universe as we transformed it toward its future. Sadly, this never happens. However, it is our goal to get as close to this ideal scenario as possible.

Real-time is greater than in-game time. This is the most difficult situation, as the in-game time is too slow now. This usually happens if the in-game universe reaches a certain complexity. As it can't be expected that the in-game-universe will suddenly become less complex, the next transformations will also be too slow, and the player will experience a slow-down effect. There are several techniques on how to deal with a complex scenario. Parallelization will be discussed soon.

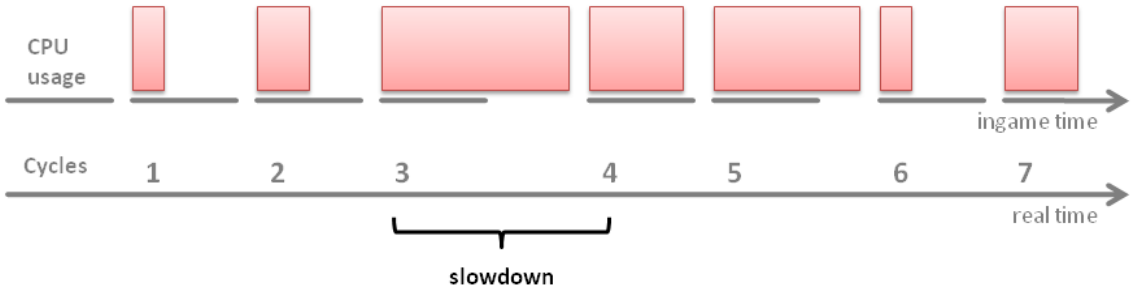


Figure 3: Constant Cycle Length

The implementation of a constant cycle length is usually a loop with a mechanic to measure the elapsed real-time (a whole topic of itself in multithreaded environments). At the end of the loop, the elapsed in-game time and elapsed real time are compared. If the elapsed real time is smaller, the loop waits for a bit. If the in-game time is smaller, it might simply continue, hoping to get faster soon, or it might push some methods to reduce precision, yet speed up calculations in the next cycles in order to catch up with real time again. Note that this "processing control" causes time acceleration/deceleration effects which might feel awkward to a player. Most games utilizing constant cycle length accept the slow-down effect to at least provide the feeling of a uniform time scale.

A good example for a constant cycle length game is the Strategy Game Supreme Commander (21). Supreme Commander (21) does a real physics simulation for all warheads and vehicles. Unlike many games, the missiles do not already know if they hit at the moment they were launched, but they use real ballistic characteristics for targeting. Therefore, a constant precision for their physics simulation is very important, and that's why a constant cycle length should be preferred. This leads to the result that a multiplayer game of 45 in-game minutes can easily fill a whole evening.

4.3.4. Variable Cycle Length

Another possibility is to use the real-time it took to calculate the last cycle as the in-game time step for the next cycle. This ensures that the in-game time will always proceed at the same speed (some minor noise might occur, but that's negligible). On the other hand, if a universe becomes complex and the real-time required for a cycle update becomes very long, a cascading effect will occur. This will increase the cycle length, as more and more time has to be processed within one cycle, which will of course take even more real time. Therefore, a variable-cycle-length system will have to supply features to reduce accuracy of calculations or even skip them, in order to return to a usable cycle rate.

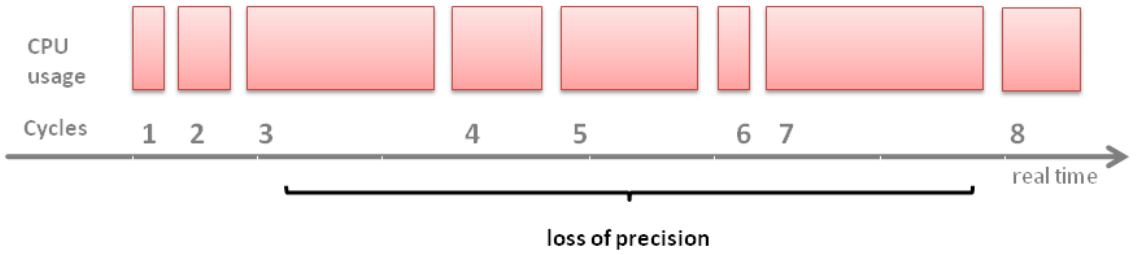


Figure 4: Variable Cycle Length

Figure 4 shows the problem of reduced cycles per second. Another problem can be seen here, too. The changes between cycles might be a bit extreme for the average computer game, but it serves the purpose for this example. If we take a look at Cycle 3, it becomes very long and exceeds the acceptable threshold. An accuracy reduction pops in, for example by telling Render Objects to ignore certain events. Cycle 4 becomes much faster now again, so the rendered objects are told to proceed as normal. This leads to cycle 5 becoming quite long again. This ping-pong behavior can lead to a lot of overhead. A certain delay should be put in place, for example by introducing a policy that an accuracy level can only be changed once every five seconds or via a similar mechanic.

4.3.5. Prediction and Continuous Cycles

If a single cycle, no matter which cycle model is chosen, gets very long, a certain stuttering might become visible to the user. Everyone who plays games every now and then will probably have experienced this, especially if trying to play a modern game on an older machine, where speed assumptions for processing during a cycle are not matched by the hardware. This issue is often ignored, but some games deploy a technology to soften the stuttering, especially if the game developers expect to have very long cycles at peak times due to highly variable universe complexity. This is a typical scenario for real-time Strategy Games.

The solution to this problem is called continuous cycles, although this name is rather misleading, as the cycles are still teleporting from one point of time into the next. Rather, a better name for this model is prediction. The idea is not to think of an entity's state as a value, but as a function. Taking the past entity states into account, a prediction algorithm might interpolate potential next states at points of time between the cycles. This allows a renderer - or a similar external module - to gain information at the sub-cycle level. As soon as a new state is ready, the prediction states are discarded and a new prediction starts for the new state.

This is of course not implementable for every kind of attribute a state might consist of. How should a prediction algorithm predict the changes of a lambda function? However, that's not required. The interface a renderer has to access is rather small, such as an object's transformation, as well as some rendered information or physical information like velocities. Likewise, the transformation-matrix and velocities are easily interpolated by linear functions. In this way the Game Objects still transform, even if the cycles get very long. This can easily be observed in modern MMOGs, where a cycle requires network communication with the game server and therefore can last for a second or two with bad connections. During these "lags", one can observe the Game Objects - like players - still moving in their last set direction and instantly be reset once a new state for them arrives. Prediction usually follows the constraint: "predict what is possible, ignore what is not". During architecture

design it should be kept in mind that prediction takes time and processing, even if just doing some simple linear interpolation, such as using the last set velocity.

As the functionality to implement variable accuracy for calculation is difficult, especially if taking multiplayer synchrony into account, there are only very few games that utilize this approach. The overwhelming majority of games use constant cycle length. Nonetheless, a variable cycle length can be combined with a constant cycle length system by running shorter prediction cycles - refreshing the state attributes according to their current prediction velocities while the "real" main cycle is still running. This way it's even possible to achieve a good prediction, without forcing the external parts, like the renderer, to know about the velocities. This is a good approach to creating a uniform and simple API for the external devices, as well as offering the option to turn prediction off on a client if the extra performance should be required.

4.4. Parallelization

An important aspect of a modern game engine is to offer possibilities for parallelization to speed up calculations. Multi-core processors are very common, and with general purpose graphic cards appearing at the horizon for commodity PCs, parallel data structures will become more and more important. Game engines could utilize parallelization by executing the updates during a cycle on a Thread Pool or a similar structure. This is referred to as "parallelization per data", as all processing is done for each data block (for example a Game Object). It is a relatively common and easy-to-implement approach, as it is very intuitive. If you are given the job to paint 10 walls, you will probably paint each, one by one. On the other hand, you might get a friend to paint 5 of them while you do the other five, which is data parallelization. However, this simple and powerful concept has some drawbacks, too. We will encounter them when talking about synchrony. Another common pattern for parallelization is "parallelization per function", which means that all objects are processed sequentially, but at the same time by different workers performing different actions. This solves some of the synchrony problems, but might cause new ones with locking. A game architecture might encapsulate the data into different functional domains in order to support "parallelization per function". We will see this approach in action in part 2 of this document.

4.5. Causality

Causality is another aspect of our known universe, which seems rather simple to us. If you are given two events, A and B, with the information that both are linked via a causal relationship, and that A happens first, it is easy to tell that A invoked B, or that B is the reaction to A. That's because causality is mapped closely onto time. Essentially, time is just a virtual currency for causal relations, introduced to make talking about time easier. Consider an hourglass for example. Measurement via an hourglass is very closely linked to the causal chains involved in letting the sand tickle through (our default watch is of course linked to causality in the very same way. An hourglass is just a very easily understood mechanism).

As a game engine already defines a new construct for time (discrete steps rather than a continuous flow) it is quite straightforward that we will also have to define an in-game version of causality. To understand the details, I will have to make a few assumptions that we will look at more closely in later chapters. Let's define that the information of an in-game entity is summed up into a structure called a Game Object. During a cycle, all Game Objects are called by an update function, which transforms them into the next (in-game) temporal state. If events with causal dependencies are

happening in a very short time frame, it is possible that both the action and the reaction happen between two in-game points of time. This means that both the action and the reaction have to be processed in the same cycle. Consider the impact of a missile for example. The action (its impact) is rapidly followed by its reaction (a shockwave), which will rip apart the wall of a building. The update function will carry over to the Game Objects and push their update functions. The missile gets updated and notes down that the shockwave has to spawn. The shockwave gets updated and notes down that the wall should be destroyed and finally the wall will get updated and be pulverized. However, this chain becomes lost if the wall gets its update message first. This would require it to repeat the same update multiple times per cycle, which is unacceptable due to pure performance reasons.

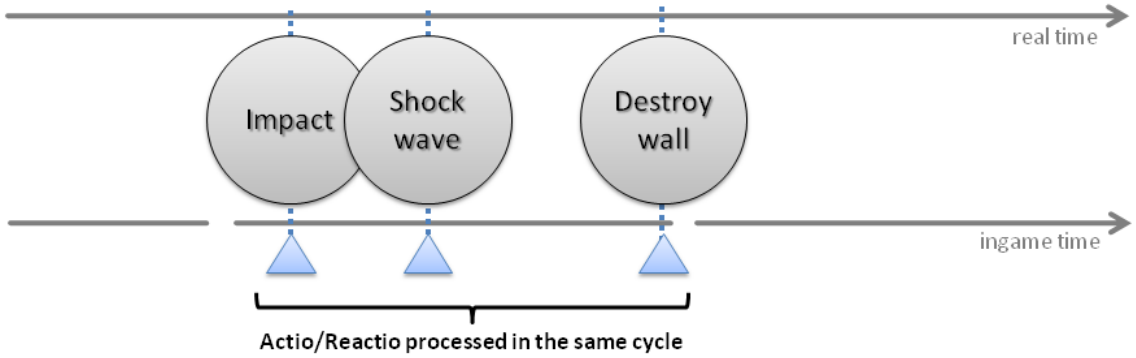


Figure 5: Causal conflicts during a cycle

The solution is quite simple. A game engine does not utilize causality as the real universe would. In-game causality is usually defined as the flow of information from one game entity to another. The sender is obviously the action and the recipient is the reaction. Note that this information is completely unlinked to time. This enables parallel processing by blurring causality through time. To use the missile example from above, once the missile gets updated, it leaves a message to spawn a shockwave. During this cycle, the wall can be updated at any time. At the beginning of the next cycle, all messages are being processed and the shockwave is spawned. All Game Objects are now updated and the shockwave leaves a message to destroy the wall. Again, the wall can be updated at any time during this cycle, too. During the next cycle, the wall will finally destroy itself as a result of the message. Using this definition of causality, the engine can freely use parallel processing for Game Objects with a relatively manageable processing overhead for causality (due to the message processing involved). The drawback is a certain delay compared to real-world causality chains. But as a game engine usually attempts to achieve a large number of cycles per second, it should be possible to get enough causal updates per second to avoid blurring events within the magical 120 cycles per second (the amount required to have a new state ready on every frame rendered by a renderer on a default 60hz LCD display). A game engine usually processes several hundred - if not thousand - cycles per second (that's of course just a rule of thumb and is heavily influenced by the architecture).

5. Multiplayer

It is almost unbelievable, but there are still games shipped in the year 2010 without multiplayer capability. That's due to the fact that multiplayer is a quite difficult problem to solve without a giant impact on game engine architecture. It will touch almost every fundamental structure, depending on the way you want to achieve synchrony. However, I'm getting ahead of myself again. Let's first think about what multiplayer actually requires.

Multiplayer involves offering the capability for multiple players to act within the same game world. In the simplest case, all players are using the same device, or in other words, playing on the same computer. That's a common approach for classic console games, where all players use the same console and just different controllers. This scenario is rather trivial for a game engine, as it just involves processing multiple inputs and associating multiple players to different Game Objects.

If players use different devices that are connected via LAN or Internet, the topic becomes more difficult. In this case, each device runs its own game instance. Basically, the problem boils down to synchronize several universes. Obviously, there is no (known) real-world picture for this type of problem. Besides the naïve approach of transferring the complete universe state from one client to all others, all clients have to achieve a certain degree of synchrony. This is in order to produce at least similar universe states, to achieve the illusion that all clients are in the same game world.

There are many patterns on how to achieve synchrony. A straightforward one is to collect the updates on Game Object states in some kind of record structure, often referred to as a cycle report. A host client could now transfer its cycle report to all clients. Once they execute it, all will share the same state. A cycle report has a relatively large transfer overhead, but requires no severe extra constraints to take into account.

Strategy Games usually feature vast amounts of Game Objects, which have to be kept active at the same time. Therefore, a lot of Game Object state updates might occur. Character-based games might even be sufficiently synchronized by transferring Game Object state updates, as players usually act quite closely with the world, and the keyboard and mouse input already maps closely to Game Object state updates. For instance, consider pressing the "move forward" button, which will map directly into a transformation or at least a translation change of the player Game Object in the default First Person Shooter. Omnipresent games will usually die from lag if using this synchronization approach. Consider pressing the mouse button while having a battalion of 100 tanks selected. This is a very common operation in daily gameplay, which will order all tanks to move to the clicked location. This invokes setting new target coordinates for a path-finding AI, which then involves a lot of processing and a ton of transformation updates on all of the selected units.

In order to achieve better network usage for these scenarios, one might ask the question "okay, so where do the Game Object state updates come from?", and that's exactly the right question to ask! They of course do not pop up out of thin air. They are caused by a function call on the respective Game Object (let's ignore dirty public fields for a moment). Considering the "move command" example from above, we could reduce the bandwidth usage by at least half (if setting a new command only invokes one additional state change: setting the new transformation. It usually invokes a lot more of them). The solution would be to note down all called functions with their

arguments, instead of the actual Game Object changes. However, by doing so, we have to take a look at some extra constraints:

If we utilize parallel processing (what we will always try to do) we also have to get a certain order into the function calls. This could be achieved by accessing the global in-game time. All Game Objects give tokens away to each function call and note down their order in the cycle report. Once the in-game time changes, the tokens are reset. Yet this still requires an additional requirement: that all functions are at least "slightly" side effect free. This means that if operation B is started while operation A is still running on the very same Game Object, the result of A and B is still the same as if they were running in sequential order. A and B could still produce side effects for future calls of their respective functions, or for a function C, which is always executed after A and B is completed (one could think of a cleanup step), but their side effects may not influence one another.

While this new constraint seems rather easy to handle, it usually causes quite a bit of thinking in some tricky cases. Still, it is usually quite easily implementable. However, this synchronization pattern has gotten us into another, much more severe, constraint: the function calls have to produce the same result on all clients. This is of course linked to the side effects problem described earlier, but there are also functions that have built-in side effects to produce their desired results. Variable missile damage by utilizing a random() call comes into mind. Random() is often implemented by defining a random host which distributes an array of random variables to all clients, of which the random() results are then chosen.

5.1. Synchrony via Domains

Do we also have to synchronize pure getter functions? At first glance, one might state "no, of course not, they don't change the state of an object". On the other hand, they are closely linked to resolving the causality. Consider a fast car collision in a racing game for example. Car A and B are about to collide. Client 1 updates car A, which is about to ram car B's side. During movement, the physics algorithm (more about that later) is triggered. It gathers the current transformations of other Game Objects and finds out that car A and B just had an accident. Another Client 2 might update car B first. Car B moved a bit forward and then updates car A. Car A now actually misses car B! Client 2 reports no collision and both cars continue to move.

This happens because a dynamic universe (exactly what our game engine is there to simulate) always works with side effects. As long as objects just update their internal state - meaning they just change internal values without asking for external information from other entities - everything is fine. As soon as an object accesses any foreign information (either a global construct or another Game Object) there is a good chance that causality is involved, which has to be taken into account to achieve virtual synchrony. There are few solutions to this problem. The most straightforward one probably is to try to reconstruct the precise update "path" a host took on all clients. This is done by also transferring meta information, such as the order of Game Object updates. If side effects occur, this solution hopes that the client universe is in the exact same state that the host universe was in and that therefore the side effect will result in the same return value. The order of access to fields or Game Objects is of course not the easiest thing to identify in a multithreaded environment. In this case, one would even have to do per-thread-call lists, which becomes very tricky if we are not working on homogeny hardware. This solution performs far better if parallelization is split by functionality, and mechanisms (or at least policies) assure that the different functionality domains

(damage, physics, input, ..) are not linked via side effects, but a precise messaging system, which can be serialized much better. In this way, each functional domain can be viewed as a single-threaded system, where finding the order of access is no problem at all. Thus execution of a cycle report containing the call order for each functional domain can be executed on any number of worker threads, closely mapping the given hardware, as long as here is only one worker assigned per functional domain.

5.2. Synchrony via Lambdas

Another solution is to also serialize the passed arguments a host receives as it called the function. If a mechanic (for example, per definition of the scripting language the Game Objects were written in) or a policy defines that all possible side effects of a Game Objects functions must be passed via parameters, the possibility is given again to process them parallel without limitation. This approach is closely linked to functional programming languages, if you see a function call as a pure lambda. The result of a (pure) lambda is only dependant on its passed parameters (47). In a real game scenario, there will always be functions that need dynamic side effects. The physics engine thus comes into mind. If it attempts to find potential collisions, it has to at least check the nearby Game Objects for their transformations. If this were mapped into arguments, then either the complete Game Objects collection has to be passed, or preprocessing has to be done to find potential collision partners. This has to be done no matter if the actual collision check would have been necessary, as the physics function call might find out that the current relative velocities are positive – that all objects move away from each other.

If the potential client hardware is very heterogeneous, there might be inaccuracy calculating values which could lead to an asynchronous state. That's for example true if one client uses single precision floats, while another one uses double precision floats. Even if the complete synchronization was resolved successfully, they will still end up in different states. Therefore, most engines utilizing this pattern are rather strict in defining the processing parameters of even the smallest engine routines. This can be seen as additional constraints added to the engine as a whole, which makes porting harder.

Considering a working on-function-call update pattern (we will take a closer look at its implementation later), we have a very powerful utility to do character-based games like First Person Shooters. Even complex actions in these environments map quite nicely on function calls. Common interaction, like moving, firing a weapon, using a device or talking to a non-player-entity, perfectly matches a few or even a single function call. If we take a look at the default Strategy Game, the bandwidth usage might still not be sufficient. A single button click (giving a movement order) still might cause a lot of function calls, as an order is an abstract construct itself and has to be converted into actual state changes (and their specific function calls) via helper functions. Assigning a new waypoint, for example, involves a path finding script, often part of an assistance-AI, to parse it. During the process the assistance AI has to ask for positions of other Game Objects for collision avoidance, which will produce a lot of management overhead, as we found out earlier.

5.3. Synchrony via Reconstruction

Are Strategy Games forced to do one of these two patterns? Nope, of course not. Games with hundreds of mechs closing in on hostile bases would have a hard time to accomplish this. Let's just follow the same pattern as how we determined the last solution to reduce network usage: where do

the function calls originate from? Or better said, what is the causal trigger for Game Object function calls? In most cases, if one would make a heuristic analysis, the answer would probably be, "other function calls ", which is not a solution that will give us more information. As a consequence, let's zoom out a bit and take a look at the whole in-game universe again. If we ignore random functions for a minute (as explained earlier, they are relatively easily solved for a synchrony scenario), a game world simulation boils down to a video, as long as there is no user input. If we know the start state of the in-game universe (something which we should almost always know, except for some scenarios where a player joins a game round which is already running) and all the rules of a universe, we can calculate the universe state of any point of time. Player input is the only thing an engine cannot expect. From a game engine's perspective, user input is the real "random", while the random() function we would expect is just another side effect function call. So why not follow the above principle and only synchronize randomness? In other words: only transfer the user input.

While this solution sounds like a very elegant one, it causes a large amount of extra constraints. The problems are very similar to the situations described in the function-call-synchronization approach using lambdas. However, now an engine has to make sure that complete processing, beginning at a user input event up to the actual Game Object state update, is serializable and leads to the same results without transferring too much extra information. Otherwise this would spoil the original goal of reducing the amount of data to transfer. It is very difficult to summarize and describe the problems of this solution without taking a look at an example, so I will skip them for now and come to them later, as we will talk about actual game engine implementation in part 2 of this document. The Telesto engine utilizes the reconstruction approach.

Games using synchrony via reconstruction may also allow players to adjust the maximum time for transferring the input data. If connections are very slow, this creates a game without producing "lag" every few seconds, but it also increases the time before the game reacts on local input. This is another tribute to the perception of players. A player is usually satisfied if he or she is capable of playing a game which reacts with a delay, but consistently, rather than a game which gives a "bad connection" message. Such an option can be seen in the popular real-time Strategy Game StarCraft (5), which is known to be playable even on the worst connections.

6. Game Objects

We now have a basic understanding of what an in-game universe is and how its rules are described and enforced. I already used the designation "Game Object" from time to time as the name of an entity in the game world from a game mechanics view, but I never talked about what a Game Object actually is. It is time to change that.

There are at least as many definitions on what a Game Object is, or should be, as there are on game engines. In the following chapters I will summarize the most common principles of Game Object architectures. Before we start, let's try to find out what a Game Object could be. In a running in-game universe, there are several types of entities. The ones that everybody notices are objects that would also be called objects in the real world. These include parts of the game world for example: an environmental object, like a tree, or a player object, like an Orc warrior. However, there are also non-spatial entities in the world, like the representation of a player in a Strategy Game, a scripted trigger activating certain events after a condition is met or certain environmental attributes like wind.

If we consider all these different types of objects to be Game Objects, we will end up with a lot of usage scenarios for them. There are, in fact, a lot of ways to implement all the logical and spatial constraints that could be represented as Game Objects. Some engines map everything on a universal Game Object structure, while others use different mechanics for spatial and non-spatial objects. Wind, for example, could simply be mapped into a physics engine constraint with an API to configure the state. No matter which approach is being chosen, Game Objects always invoke a complex architecture.

6.1. Simple Information Models

The most common solution to implement Game Objects is to map them on classes, just like you would in an office program. If you are working on a database tool to manage employees of companies (probably the most common example, right after "hello world"), you would think about what defines an employee and create a class corresponding to your findings. You will probably create a "person" class first, from which the employee is inherited. Why not follow the same pattern for soldiers, guns, and tanks in a game? There are good reasons to use this approach, and good reasons to think of something else. It all boils down to the question, "what do we need?", again. There are in fact modern games that use this technique, and these models can grow far beyond anything one would describe as "simple".

The benefits of a simple information model can be found within the straightforward mapping of objects onto classes or instances of classes. That's exactly what a developer should be used to and inheritance is a very powerful tool to map relations. Yet, just like in a growing office program, there are difficulties linked to inheritance, too. If there are multiple "is-a" relations on a single object, multi-inheritance might be required, which tends to lead to a lot of copy and paste code, as it might have to be mapped on interfaces (if you are working in a language which supports multi-inheritance, the Simple Information Models become more attractive even for complex scenarios). On the other hand, if the Game Objects are relatively unique and don't overlap too much in terms of their functionality, this approach might still be useful. That's often true for First person Shooters, where each character has its unique characteristics and behaviors.

6.2. Actor Models and Active Objects

Once performance tests are running on a game engine, it becomes obvious that there is a fixed overhead per Game Object. The cause of this overhead is the update-call that all Game Objects have to receive during each cycle, as the cycle-routine cannot know if the Game Object wants to do something. The idea of actor models - better called active objects in the context of game development - is to avoid this overhead by only updating objects which are currently doing something, hence the name “active objects”. To achieve this goal, we will have to move the time and causality loop from the global universe instance to each Game Object. Once an event is triggered, the Game Objects will start a private simulation for themselves, refreshing their state. Other objects may ask a Game Object for its specific state at a certain time and the Game Object will return it to them as soon as it is available. The asynchronous "as soon as available" implies using a message system, which is exactly what most active objects model do.

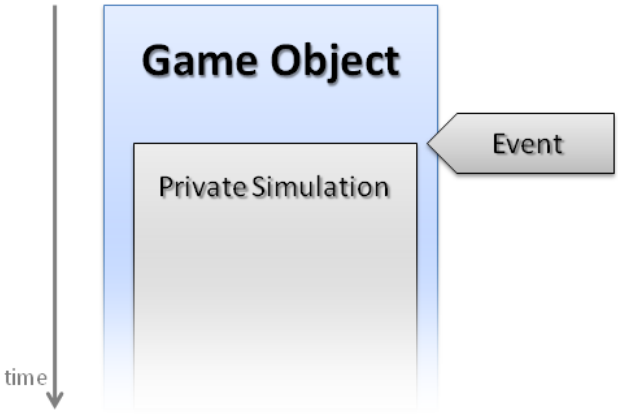


Figure 6: Active Object with a private simulation

The ability to manage the actual state-transformation-over-time simulation per Game Object offers many interesting features. This approach is often used for scientific simulations, like a physics or chemistry model, as it allows the actual behavior over time in the specific Game Object to be directly overwritten, rather than relying on a "god instance" - the cycle update runtime. This is of course much closer to the real behavior of things. Consider writing a simple chemistry simulation. The scientist is able to program the complete behavior of an H_2O molecule within the same code block and does not have to wrap it in different on-collision, on-update, or on-reaction handlers. It also allows a much better optimization for precision, as the complexity of the private simulation can be adjusted per Game Object. A simple rigid body object could just refresh a transformation, while a fluid simulation model Game Object might host a complete mini-game-universe with sub-Game Objects representing particles. Active objects models are often used in scientific simulation, but only very rarely utilized for real time applications. However, why is that, given that they seem rather attractive?

The problem can be found in the fact that active objects do not share temporal coherence. This means that at a given point in (real) time, all Game Objects within the in-game universe are of different in-game time states. This results in some severe problems if objects are interacting with each other. Consider two active Game Objects, A and B, which both receive an in-game event at the same point of in-game time (t_{g0}) and begin their private simulation. One (A) is a rigid body moving

through space, the other (B) is a super realistic liquid simulation. It's obvious that in-game time will pass much faster for object A then it will for object B. Let's now have object A attempt to read a value from object B at in-game time t_{g1}. Let's call the real time point when object A reaches the in-game time, t_{r1}. At t_{r1} the state of object B is somewhere between t_{g0} and t_{g1}. Object A of course wants to receive the state of the accessed variable at t_{g1}. Therefore Object A will have to halt its simulation until object B reaches t_{g1}. During this time, all objects that might want to access object A will also have to wait. In an extreme case, this will lead to all objects being processed sequentially, sorted by their causality, which is extremely slow.

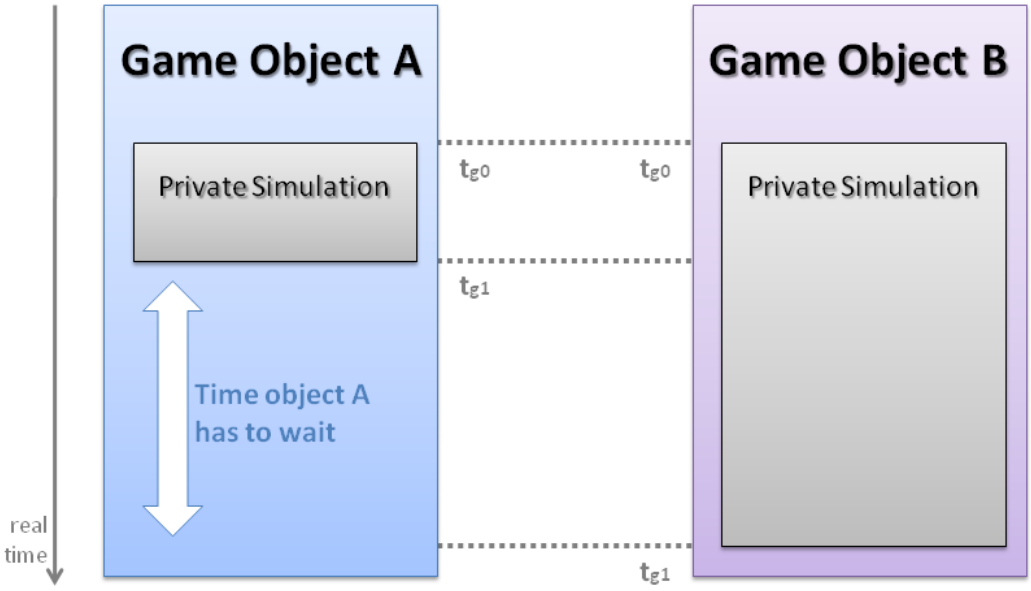


Figure 7: Active Object dependency problem

Similar and even more severe problems arise, because the in-game time resolutions of different map objects vary. Therefore it's possible that Game Object A will request a certain time state that Game Object B never actually produced, even though both are advancing through time with the same speed. These scenarios have to be handled via policies like interpolation, snapping to the closest point of time, or similar constraints. In the worst possible case, active objects always require a lot of caching, as all Game Objects have to remember all past states they produced.

The solution to at least partially solve these issues is to introduce a global clock mechanism, which determines a minimum time. This means that all Game Objects are at least within the same time window. All will halt until the slowest reaches the minimum time. By doing so, the chance for large sequential chains is reduced and all states before the minimum time can be removed from the cache.

A global clock that determines a certain time for all entities - doesn't that sound familiar? That's exactly what a global cycle is. Likewise, we have even reforged the Update() method on all Game Objects, only now it is a method that accepts the global minimum time and reacts on it. At the end of the day, active objects give no real gains to game engines, which is why they are never used. The benefit of active objects is the adjustable precision and the way they can be programmed. That's why they are often used for scientific simulations. Nonetheless these simulations aren't real-time applications. They produce a certain end state, which is a result or a video, but can take any amount

of time to do so. A game cares less for precision however, but requires a real time environment where actions of a user are processed as fast as possible. Actor models and active objects aren't bad; they are just not useful for creating an interactive real-time universe.

6.3. Modular Shared Memory Models

The desire to have a universal Game Object and optional modules that define what the Game Object actually does is very attractive for developers. This allows a lot of salvage, since in the best case a certain module might be reused for every instance of its application. For example consider a render module which manages the rendering of an object. If this renderer module could be reused for every Game Object, it would save a lot of work. A simple data model might inherit from a base object with a render() function, but multi-inheritance might cause the render() method to be implemented several times via copy and paste.

In reality, there will be object specific information which needs to be taken into account. The renderer module example from above might require information about the actual model to render, special effects or even dynamic structures like animation scripts. These parameters are often called arguments of a module or configuration data of a module. Besides the different names, these constructs strongly resemble lambdas, as you find them in a functional programming language. Furthermore, that's actually the approach for shared memory models, where the Game Object just supplies formatted memory - for example in the form of blocks used for the atoms of a functional programming language - for optional modules assigned to a Game Object. A module manages its part of memory of its own free will.

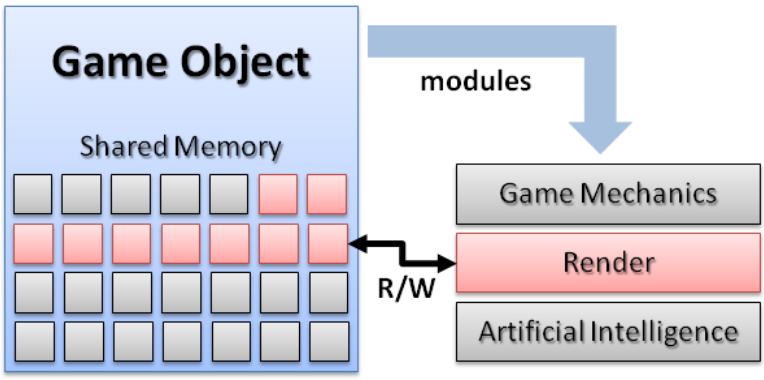


Figure 8: Modular Shared Memory Model

This approach of course requires a very tightly integrated scripting language. Many runtimes don't offer sufficient support to isolate single atoms and manage them, so the runtime will probably have to be developed along with the engine. A slightly different model is to move from a global variable storage, where, in theory, a Game Object could access every attribute associated with it, to a model where instances of the modules are stored per Game Object and each of these creates its own memory. This works well with third party runtimes, as it simply requires the creation and maintenance of a script block or environment instance, which is rather common.

Different modules are combined by simply adding them to the Game Object and giving them write access to shared memory blocks - for example by supplying them variables to write to in the Game Object's script environment. In a real game engine, most modules are interconnected. The renderer

module, for example, requires transformations which might be produced by a physics module. To access them, the render module just asks the Game Object if a certain value is available. Unfortunately, this creates a new problem: the modules have to know about each other or have to ask the Game Object if the required modules are present or at least if the required values are managed by any other module. This circumstance either reduces performance, as more checks have to be made at runtime, or reduces the reuse of modules, as they might have to be adjusted in respect to different combinations. For example, this could be whether the transformations of a Game Object are set by a physics engine, with additional acceleration information for the renderer, or by a game mechanics module, which might just set the transformation depending on certain conditions.

Yet, the goal of a uniform Game Object without inheritance is achieved and the shared memory access is almost as fast as a direct field access used by simple data models. It boils down to replacing a GetHealth() function with a GetValue(name) function, thus accessing an internal environment of a scripting language. However, it also allows direct evaluation of scripts on the Game Object environment. This allows shifting a lot of functionality into the scripting language without further effort. I will demonstrate how an implementation of a modular shared memory model might look like in the second part of this document.

6.4. Behaviors

The modular shared memory method already offers a relatively large potential for reuse of code. Nevertheless, there are games where almost every Game Object requires very different combinations of modules or highly situational scripts which tend to create interconnections between modules. Namely, First Person Shooters. Consider the level of a First Person Shooter's single player campaign. Most enemies might be instances of a few classes, for example a default hostile soldier. However, they will have very distinctive AI parameters. Some are positioned as snipers on a roof and others rush towards the player. Later on walls are destroyed to open up new passages and other scripted events take place, which override the actual physics engine for a more dramatic, handcrafted effect. The ability to combine any modules during level development at will increases the usability and development speed. That's the idea behind the behavior pattern.

The modules are renamed into behaviors and are no longer assigned memory blocks. They instead store their private attributes in their Game Object specific instances. To do so, they might use an environment of a scripting language or, if there is a known amount of data to be stored, they will store them directly in private variables. That's completely up to the behavior´s implementation. A behavior only knows its associated Game Object. If information or interaction with another behavior is required, the behavior sends a message to the Game Object, which will serve as a message hub and redirect the message to modules which can answer the message. To do so, each behavior comes with a capability description, which tells the Game Objects which message types it may deal with.

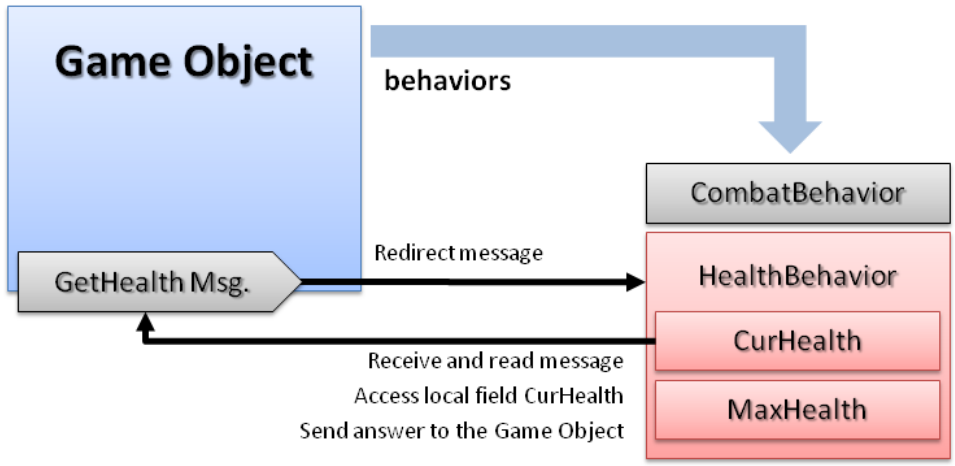


Figure 9: Behaviors

A message system with message queues produces an overhead. A request first has to be packaged into a message and then be sent to the Game Object and redirected to the proper modules. This overhead is significantly larger compared to shared memory block access. However, the shorter time-to-market and easier debugging of behaviors is often worth the performance loss, especially if the game genre is not too dependent on the update speed of Game Objects and is instead limited by other factors, like the speed of a renderer. Again, First Person Shooters are a good example, while Strategy Games require very fast update routines and usually work with classifications even in their game design (an Abrams Tank for example) which leads to a lot of Game Object instances with the exact same module configurations during runtime.

Another benefit of the behavior model is that it is a very good opportunity for integrating multiplayer synchrony. The messages already perfectly match the requirements of a synchrony with lambdas pattern, as the messages are nothing but function calls. An agent could be implemented in the Game Object, collecting relevant messages, ordering them following a given pattern (multithreading has to be taken into account here) and serializing them into a report. Synchrony would require additional processing and overhead for simple data or shared memory models.

6.5. Game Object Containers

We now have a rather good overview on what a Game Object could be and how it could behave. However, we have still left the question open as to where and how the Game Objects are actually stored. In most engines, there will be a global Game Object container storing all of them, with different functions to receive Game Objects from it. The easiest implementation would be a simple array, but that's rather suboptimal. Instead, the containers should try to use an optimized structure to accelerate the most common calls. The implementation of the Game Object container is therefore heavily influenced by the gameplay, even more so than the Game Objects. For example, a racing game will probably have many calls relating to the position of a Game Object, such as collision checks. Therefore, a spatial container like an octree will yield the best performance. A Strategy Game, on the other hand, might produce many calls relating to the logical characteristics of an entity; for example the units of a certain player. A hierarchical tree structure representing relations between the game mechanics attributes of entities might be better suited for this task. I will not go into the depth of how octrees, hierarchical structures or grids are implemented. There is a lot of

other literature about these subjects and their implementations are not directly related to game development itself.

Part 2

Telesto Game Engine

7. Telesto

I called my architecture and implementation of a game engine “Telesto”. Or better said, Telesto was the prototype that survived. Due to the large spectrum of components a game engine consists of - we discussed the important modules in part 1 - it was impossible to craft efficient plans or UML diagrams. I therefore chose a more direct way. I created a rough design plan with as much detail as necessary to get an overview of the interconnections and APIs. In most cases, such an overview diagram already uncovered architectural flaws or issues. If the plan seemed alright, I started implementing a rough prototype. If the prototype was still not to my likening, I went back to the sketch board and started over again. Of course, this might not have been the most efficient method, but it offered some protection against severe architectural flaws that had been missed and could kill your project once half way through. I nicknamed each architectural prototype (that was different enough from its predecessor) after one of Saturn's moons. As Telesto is definitely neither the most popular nor the largest moon, you can guess that many redo cycles where involved.

7.1. Motivation

The idea to create a whole engine from scratch arose a few years ago. I'm working on a group hobby project, called Fleet Operations (18). An extensive mod (called a total conversion by the community) of the Star Trek (48) real-time Strategy Game Star Trek: Armada II (19). During development, I came into contact with a lot of architectural features. Or better said, architectural flaws, as you only notice the architecture if it denies you from doing something. As Star Trek: Armada II is aging and quite a bit away from what you would call modern, we started replacing modules, like the game mechanics, with new, self-written ones. By doing so, the idea "why not do the complete stuff?" popped up. That's basically where the idea of Telesto began. For now, the usage scenario for Telesto will be an independent small game taking place in the Fleet Operations universe.

7.2. Goals

Defining the goals of a project as precisely as possible is the first step to success. Especially in the gaming industry, where there will always be a ton of change requests. So make sure that at least your basic architectural goals are well defined. As Telesto is "just" an engine architecture, without an actual game being developed based on it, I could define the goals quite well. This chapter will summarize them. Before we proceed, I should talk about the target usage scenario for the Telesto engine and architecture. There are of course plans and relatively detailed gameplay documents for games I want to develop on the Telesto engine, yet their development will start once the engine is finished (or at least as finished enough to start development) and are not part of this project itself. This protects me from too many change requests in regard to gameplay and game mechanics during engine development. In a commercial project, game and engine development would proceed at the same time. For instance, game mechanics prototypes, are often developed on an older engine - if available - while the actual engine for the respective game is still in development. This can be seen in an early image from the StarCraft (5) alpha, which was based on the WarCraft (49) engine, before the actual engine was used for the final version, resulting in a complete redo of both the game mechanics and the content scripts.

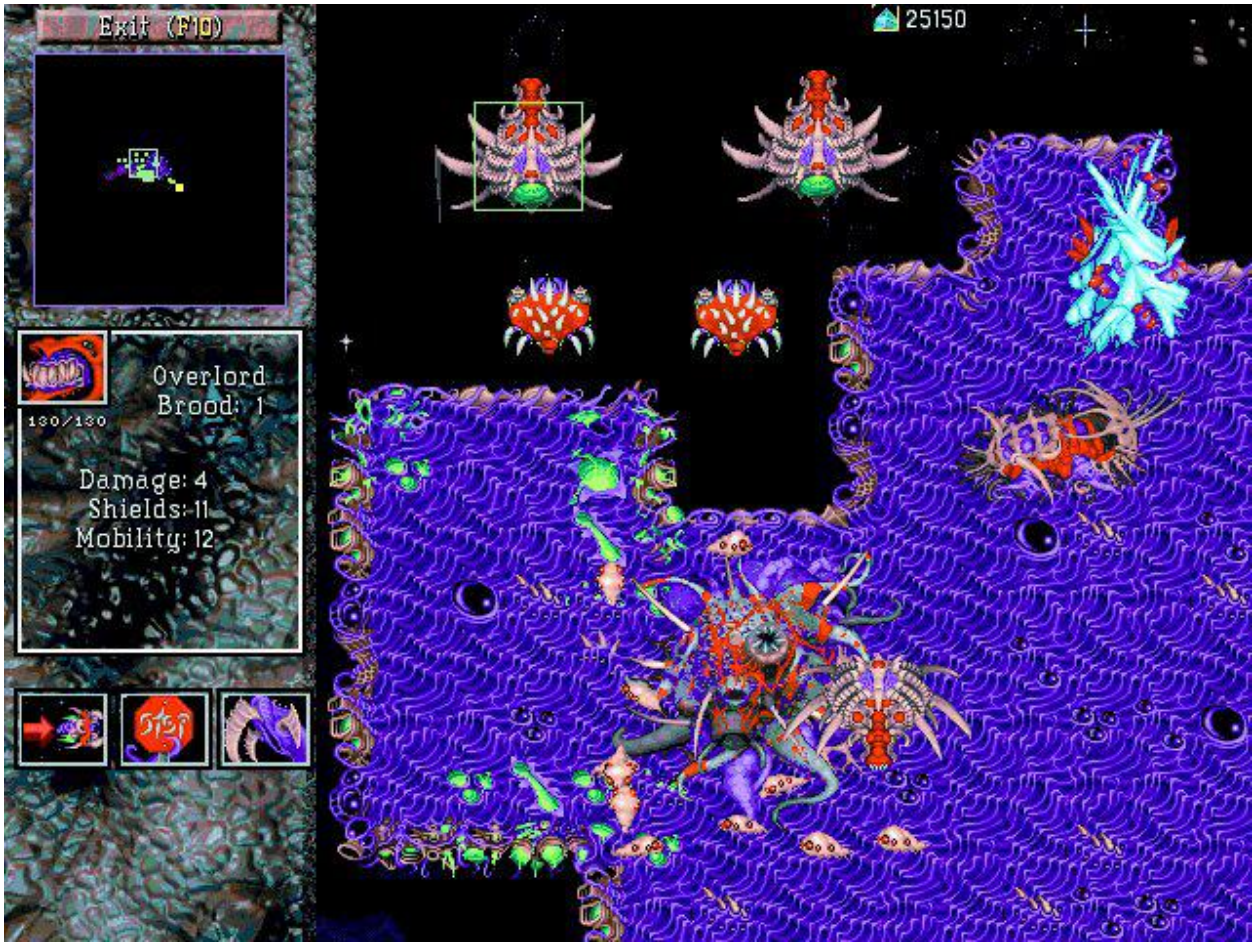


Figure 10: Early StarCraft alpha version, Blizzard Entertainment

The target metagenre which Telesto should be able to handle is an omnipresence-based game. More precisely, the target genres are space-based real-time-Strategy Games and space-based MMOGs. The first project based on Telesto will most probably be a space-based MMOG hybrid with a large portion of strategy content mixed-in. This leads to a few critical core requirements, of which the most important are to achieve an efficient handling of a large amount of Game Objects from an omnipresence-based perspective (where no sphere of influence can be expected) and to implement a highly efficient network code for low-bandwidth multiplayer.

7.2.1. No external dependencies

The goal of this project was to design a complete engine architecture. Utilizing third party software modules would have denied this, as they always bring their own requirements, which in turn would influence the overall architecture. I therefore decided to write everything from scratch, as this gives me the most freedom. Many engine developers do this too, but only because they wish to be independent from bugs in third party modules, which could delay development, or in order to achieve maximum flexibility when it comes to modifying source code for a certain unique scenario, which is always easier if you wrote the package to modify yourself (given you even get the source code of a third party module).

Developing a complete engine from scratch also offers the potential to combine modules more tightly. For example, this was true for Telesto's scripting language. The whole architecture benefited

from the low-level management options available, as LScript (the Telesto scripting language) and its runtime don't run as separate modules and have to communicate with the engine via a pipeline. This would have been necessary if they were a third party scripting language like LUA (50), but they are instead directly accessible and modifiable by various engine constructs. We will discuss more about LScript in a later chapter. Some commercial engines even implement low-level constructs like arrays or complete memory management.

The only external dependencies for Telesto are the .NET Framework, which is obviously necessary if programming in a .NET language, and SlimDX (51), an open source framework which tightly wraps DirectX (52) to be accessible from .NET applications without superfluous functionality. SlimDX (51) is rather popular for developing DirectX .NET applications since Managed DirectX was discontinued by Microsoft in favor of their game engine framework XNA (53). XNA (53) was not an option for Telesto, as it would have defined far too much of the architecture. Likewise, some characteristics of XNA, such as the graphics pipeline, did not match my ideas and requirements.

7.2.2. Full .NET Project

I am a strong believer that C++ is no longer the one and only valid programming language for high performance projects. Since I'm not a fan of the, let's say, unique syntax of C++, I was rather quick to decide that Telesto should be developed "without a single line of C". My choice was Visual Basic, based on the .NET 4.0 Framework (54), which was released shortly after I started the Telesto project. There might be some good arguments to choose .NET over other alternative languages if the goal is to avoid C++, but the truth is that I chose Visual Basic as it is my favorite language. Performance measurements show variable results when comparing .NET and C++. A rule of thumb is that a .NET application is about 80% as fast as a well written C++ application which does the same thing. Now the devil is within the little words "well written", which is extremely hard to achieve in C++ (at least for me, but most developers will agree), especially if working with a highly interconnected and multithreaded project like a game engine.

Commercial projects generally avoid tying themselves to a development framework like Microsoft's .NET. There are of course economic reasons, but also deployment difficulties. A .NET game might run well on every PC which runs the respective .NET Framework, but you can send ideas for porting it onto a console directly into the trash bin - except perhaps for a future Microsoft console. The Xbox 360 (55), for instance, least supports the .NET Compact Framework, but only a subset of the full class library is available. There are also functional differences due to the hardware characteristics of the Xbox, like limited thread functionality. Not the best prerequisites to begin porting, but well, porting a game engine is always a difficult endeavor.

7.2.3. Maintainability and flexibility

You will find maintainability in the description of every software project and flexibility in at least half of them. In the context of a game engine, maintainability means robustness in the face of various changes which are involved in maintaining a game, such as adding or changing content and especially game mechanics scripts. There are several ways to achieve this goal. In Telesto, I chose a strict divide-and-conquer pattern, by separating the different functional domains, like game mechanics and physics. As development went on, this proved to be an incredible useful approach, as it allows development of each part of the functionality at a time, while still offering precise interfaces to do testing. Otherwise it's hard to test and debug a physics module, for example,

without the rest of the engine being available. Still, complete separation was not possible, without accepting a severe performance loss. The scripting language blurs the boundaries between the different modules a bit, and there is a shared - but very lightweight - micro-core-runtime. This serves as the global universe clock and offers some generic functionality, like access to the resource manager (which itself is separated in a module, too). As long as the performance impact was not too severe, I decided to chose the most divided design, in order to achieve a high potential for modularization. If performance went down too much, I chose a faster solution. At the end of the day, we are developing a game engine: performance is king.

Flexibility is connected to maintainability, but it usually means a different aspect. Flexibility covers the ability of an engine to be adjusted, apart from the common dimensions of changes that pop up during the maintenance phase of a game. This could, for instance, be the process of integrating a new renderer or a new renderering feature. The latest 4.0 patch of World of Warcraft (2), for instance, introduces a new DirectX 11 renderer and new renderer features including improved reflection and water effects. This is a strong hint for a very flexible engine. This can be achieved via separation, just like maintainability, but there will always be some links which are hard to come by. For example, this could be the link between the resource manager and the renderer. The resource manager generally has to know details about the renderer, if it is to create and manage the resource handles for textures on the graphics card. These links were broken in Telesto by utilizing a distributed resource system, with the resource manager simply offering a shared file system. There are also several exchange formats for resources and other information. Telesto also makes strong use of descriptions. This has allowed me to identify a specific minimum functionality platform, the Telesto Foundation, with shared APIs to add external modules, like the renderer, AI or game mechanics. This offers a maximum flexibility to add next generation renderers without redoing the content, while still keeping as much performance as possible. Just as with maintainability, my credo was "performance first". More about these details in the next chapters.

7.2.4. Synchrony via reconstruction

A real-time-Strategy Game with its omnipresence-based players and a highly variable Game Object count combined with strong multiplayer dependency of an MMOG literally cries for synchrony via reconstruction. It was in fact one of my first design decisions to implement this pattern. I was quite lucky to stumble across this question so early, as it had a far greater impact on development and architecture design than I had imagined. If multiplayer is the goal of an engine, I can only recommend one to think about synchrony as soon as possible.

Synchrony via reconstruction is definitely the most difficult multiplayer pattern to implement and it caused the scrapping of more than one Telesto architecture sketch. Telesto offers a Synchrony Brush framework to implement synchronized causality. What exactly a Synchrony Brush is will be covered in a later chapter. For now let's say it is a modular structure to implement causality. Interaction and APIs for the brushes changed a large portion of the LScript API and caused many processes to become asynchronous, what caused additional management overhead. Therefore, I would not recommend to implement synchrony via reconstruction unless necessary. A First Person Shooter engine would work perfectly with a simpler system. A Strategy Game won't, as long as you want many players in the same game round (this seems to be the reason why Strategy Games are still fixated on the "eight players per game" we had a decade ago).

7.2.5. Prototype

Last but not least, Telesto is a game engine and a game needs to be played! I don't need many words to explain why a prototype was absolutely necessary. A prototype is of course an important tool to test performance and usability capabilities as well as development workflow and the efficiency of the APIs. What seems to be elegant during engine development might end up being impracticable. I had many good experiences with prototyping during the development of Fleet Operations (18).

7.3. Exclusions and options for the future

Game engines are developed by dedicated teams in two or more years and delays of even a few years are nothing uncommon. Doom 3 (56) was, according to John Carmack, originally planned for 2003, but its final release was in 2007. With this in mind, it is obvious that Telesto must be lacking something, as it was developed by a single person in about half a year. My goal was to end up with a playable prototype of the architecture, in order to begin development on a small demonstration game as soon as possible. The missing gaps are therefore optional elements or in-depth extra features which are not ultimately required. The full minimum feature spectrum of a game is implemented and running in the Telesto prototype. For a better overview, I will now list the elements I did not count as part of this project, but will include in the next month. The list will also include some optional features which are not really required, but high on my wish list.

Renderer	Developing a powerful renderer capable of up-to-date effects can take a year or more. I therefore did not include the development of a modern renderer as part of this project. The prototype of course features a basic renderer - otherwise it would be quite a dark game - but it is lacking impressive post production effects, deferred rendering and similar features. It is just a simple DirectX 11 renderer to present the engine itself. The Telesto architecture supports modular renderers. Connecting a new renderer is just a single statement. Development of a more advanced renderer will therefore not affect the rest of the engine architecture.
Artificial Intelligence	A real-time strategic artificial intelligence system is still undiscovered country. There are no real strategic AIs out there. For the scope of this project, I ignored the AI altogether, although the architecture already supports necessary interfaces, like an extractor, to achieve an AI world or input modules to connect the AI just like an human player. I will probably venture into this sector more for a separate project.
Optimization	The Telesto prototype offers the full spectrum of functionality required to create a demo game. Yet, most modules still offer room for performance optimizations. The current implementation was made for a maximum of functionality, rather than performance, as highly optimized code is hard to change or refactor. That's a common approach for commercial projects, too, where performance optimizations are often done via patches in the maintenance period of a game.

Physics Constraints	The current physics engine is a rigid-body dynamics simulation. This is quite sufficient for a space-based game, which is the usage scenario of Telesto. Yet, I want to implement a few more constraints in the next month, for better damage visualization. This includes model deformation and a combination constraint that allows one to slice through a starship's hull and separate parts.
Tooling	Development tools are an important part of an engine, as they may speed up the content creation pipeline. At the moment, the development tools for Telesto are rather simple. The scripts are edited via Notepad++ (57) and I wrote a few scripts for 3D Studio Max (58) to export geometry. I will start developing a better LScript IDE next and a more powerful 3D Studio Max (58) plug-in is also on the "to do" list.

8. The Telesto Architecture

The Telesto architecture divides the engine into three large sections: the Core, the Foundation and External modules. This was done for several reasons. First, separation brings clear interfaces. This is a great benefit during development, as it allows one to define good module tests, which are still relatively close to the real use case of a running game. This compensates for the tendency of module tests to become too abstract, especially if a highly dynamic part like a scripting language is involved, which is hard to module tests completely.

An additional, less obvious feature of a multi layer architecture is the better control of incoming change requests and feature management. Each layer acts like a kind of portal. As long as a change request does not pass a gate in its functionality, only a certain part of the application's code has to be changed, which greatly stabilizes an engine. For example, let's consider a new physics engine or that a new physics engine feature has to be implemented, such as model deformation. This is a relatively deep change. It of course has an impact on the "Externals" layer, as a renderer has to be able to draw the model deformations (model deformations are generally not stored as new models, but modifications on a deformation grid projected around the actual model). The "Foundation" layer contains specific modules of a universe, like the physics engine. This layer is obviously touched too. Kinematics will have to be changed and the collision response systems will have to create proper deformations. The "Core" layer, however, is untouched. It contains the basic runtimes, like simulating time, calling objects and offering the scripting language. Therefore, for the purpose of enhancing the physics engine, we can see the complete Core as a black box. This improves stability of the whole system, as there are larger subsections which are completely unchanged.

Similar benefits are always utilized in application development once a certain complexity is reached. Yet, they are often discovered situationally or - in the worst case - they pop up by accident. In order to produce a complex piece of software - like a game engine - in a short amount of time, it is incredibly recommendable to actually design such functional borders and modular separation during the architectural design phase. Separation also brings problems which have to be taken into account during architecture development. To continue the example from above: the physics are part of the Telesto Foundation. This means that the actual time and cycle management is unavailable to the physics engine, as that's part of the "universe runtime" located in the Core. Yet, a physics engine has to subdivide time once a collision has occurred, in order to achieve the most precise collision conditions. The physics therefore mess with time. This will have an impact on the design of other elements, like scripted physics-based animations. These animations are an element of the Foundation, too, but are not necessarily sub-dividable in their temporal behavior. The architecture or APIs have to react to these constraints and dependencies.

8.1. Visual Basic

Naturally, the chosen programming language has a large impact on the resulting architecture. Visual Basic offers some rather uncommon constructs and features which are worth summarizing to get a better feeling for the actual implementation. The most-used feature in Telesto are definitely Modules. Modules break Object Oriented development patterns, as they allow one to put functions directly under a namespace. Behind the scenes, a Module is created as a shared (the .NET notation of singleton) class and the compiler resolves references automatically. This shortens code

tremendously, as it allows one to write `Spawn(..)` instead of `Telesto.Game.GameObjectContainer.Spawn(..)`.

Unlike C#, Visual Basic was not configured for performance, but for reliability and security. A relatively intense compiler configuration was necessary in order to receive sufficient performance. The compiler configuration is not covered in this document. Microsoft has released a paper on .NET performance improvements: Improving .NET Application Performance and Scalability (59).

9. Core

The Telesto Core contains the basic functionality that keeps a universe running. The Core is rather universal and could be used for most genres. The following chapter provides an overview for its prime functions.

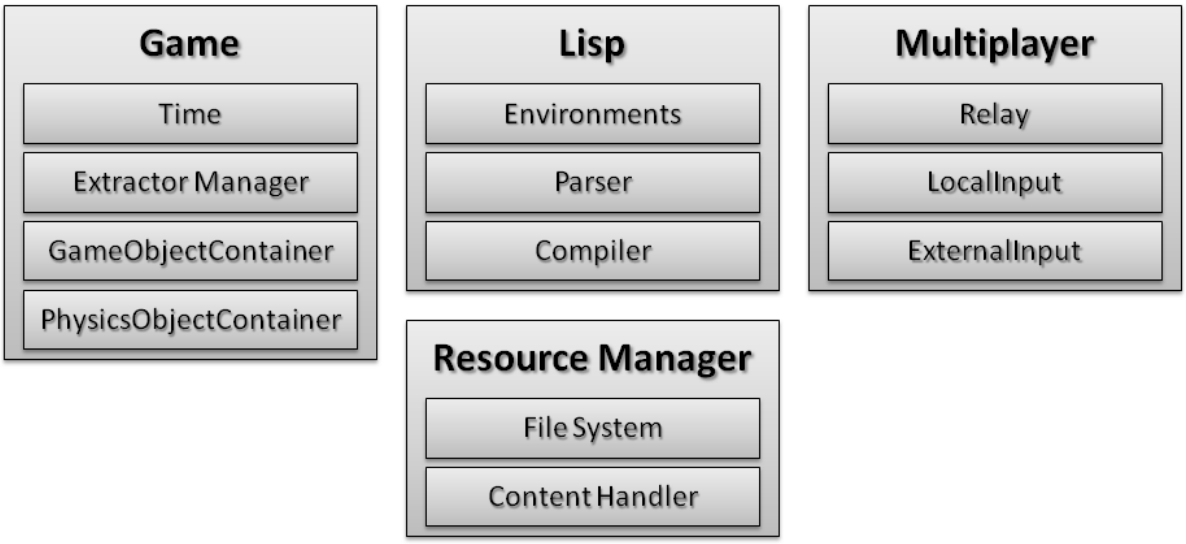


Figure 11: Telesto Core Overview (Functionality)

9.1. Time and cycles

The cycle's runtime is the core module of Telesto. It supervises the cycle management and calls modules or Causality Brushes to create temporal coherence. It also features elemental core functions like starting, pausing, and ending the game. It also has to deal with potential delays of the various subsections. In other words, the cycles, the time loop and its management structures are what actually create the in-game universe and serve as a central node for the Core, Foundation and all external modules. Without it, Telesto would just be a collection of APIs and objects to do game procedures. The cycles brings the "real-time" into Telesto.

Telesto uses layered cycles to process different functionalities in different resolutions and features a comfortable framework for cross-cycle processing. The straightforward purpose of the time loop in a game is of course to update the Game Objects. As described in part 1 of this document, there are two ways to do this: either with a constant cycle length or with a variable cycle length. As Telesto is designed for Strategy Games or MMOGs it will have to deal with massive amounts of Game Objects and a relatively complex game mechanic - shooters tend to be quite simple in this regard. A larger Game Object count makes it less feasible to integrate procedures to reduce the processing time dynamically, as a certain static overhead per Game Object is always involved. Using a variable cycle length, this could lead to overly complex scenes without a chance to solve the situation. That's why I decided to use a constant cycle length for the generic update loop.

As I have to expect very complex universe states with many upgrade routines, I decided to make the cycle length quite long compared to similar game engines. To compensate, I added a variable length

prediction cycle, which runs while the update cycle (called a Brush Cycle in Telesto, as the Causality Brushes are resolved. More about that later) is processing the next universe state. The prediction cycles transform velocities on attributes - like the translational velocity in the context of the physics engine - into states for presentation (like the renderer). This can be equated to approximating a function through linear intervals: If we see the whole universe state changing over time as a function, then the Brush Cycles draw points in a fixed distance, in order to provide causality. The prediction cycles will approximate the function between two brush-cycle-points as close as possible with the performance available. If the speed slows down, the prediction cycles will get longer and less precise. They might grow up to the length of a Brush Cycle, but a Brush Cycle cannot be missed. In addition to these two cycles, there is an additional time dimension: the Safe Cycles. These cycles are even larger than Brush Cycles and add the user input to the universe. The follow diagram shows a realistic example of the cycle duration settings as used in a Telesto demo application.

Figure 12: Telesto cycle layout

The prediction cycles are of course much smaller than the diagram above suggests. The Telesto architecture typically runs with 500 000 cycles or more (multiple millions) per second, depending on the current universe and the hardware it is running on. This allows very smooth prediction, hiding the relatively long Brush Cycles, where causality and events are processed. In fact, the Brush Cycles are for the Safe Cycles, what the prediction cycles are for the Brush Cycles. From an engine's perspective, all universe states are functions, as long as there is no user input. From an engine's perspective, the user input is the real random. This means that the Safe Cycles are actually the point at which real universe changes happen - the universe function is changed into another function, to stay with the image above. The Brush Cycles serve the purpose to get a higher resolution for causal events, which reduces the processing required to resolve causal changes which happen during a cycle. This was already discussed in part 1 of this document.

The time is measured in microseconds, using high resolution hardware devices. It is important to use a thread-safe measuring method, as the real time elapsed (or at least something close to it) is required to organize the cycles, and not just the time the main thread where the loop runs in was active. Most OS-clocks are thread-based clocks and therefore the hardware devices should be preferred. Still, an engine should be robust to fluctuations in time measurement, as even the hardware clocks have their "unique" behaviors in some situations, like jumping back in time or offering highly variable intervals. Note that only the time resolution of Telesto is in microseconds. The precision of time measurement is about half a millisecond on modern main boards. Setting up correct time measurement is the first difficult task encountered during game engine development.

9.1.1. Brush Cycles

Most of the processing of Telesto happens in the Brush Cycles, so it would be best to take a closer look at them. Safe Cycles add input handling, but this will be discussed in a dedicated topic later on. A Brush Cycle begins by checking if all simulations - for example the physics - are completed. It should not happen that a simulation takes longer than 10 milliseconds to complete - the Telesto engine is rather tolerant with a 10 millisecond cycle length. Most comparable engines require cycles of 1-2 milliseconds to achieve fluent results. Yet, the engine simply has to check if everything is done before starting the next cycle. If a simulation is still running, the engine will have to wait, slowing down the in-game time as expected.

If all simulations are complete, the engine has reached an idle state. This is the very short time window where synchronous operations can be applied. That's right where the Causality Brushes are fired. Generally speaking, the Causality Brushes apply the results of the simulations on the entities with respect to defined causal constraints. Obviously, this has to be done in a synchronized window, to assure that network synchrony can be achieved. If other threads access entities while changes are applied, it is impossible to achieve the same result on multiple clients without significant overhead. We will cover more about Causality Brushes and multiplayer synchrony in a later chapter. Once the synchronized window is done and all synchronized operations are completed, the simulations are begun again to produce the results for the next cycle, and thus the Brush Cycle is finished. The following snippet shows a very simple and cut-down example of a Brush Cycle's implementation in Visual Basic .NET.

```
1  While Not PhysicsObjects.SimulationIsDone
2     Console.WriteLine("Waiting for physics!") 'ToDo
3  End While
4  While Not GameObjects.UpdateIsDone
5     Console.WriteLine("Waiting for mechanics!") 'ToDo
6  End While
7  PhysicsBrush.Paint()
8  SpawnBrush.Paint()
9  'Start the next cycle:
10 PhysicsObjects.Simulate()
11 GameObjects.Update()
```

This example uses two brushes, enabling physical interaction, such as collision and spawning new entities. The delay handling for slow simulations has been skipped. Depending on the way in which the cycle loop is designed, this could either be done by halting the in-game time measurement until everything is completed or by waiting in a loop, while the in-game time continues, just like in the example above. As the hardware devices can't be halted, the first option requires a complex handling of the in-game time. This could be done by accumulating an in-game time offset which is increased by the time the game spent waiting on the simulations during each cycle, but this could lead to jumping timestamps for prediction cycles. The second option features the potential problem that a Brush Cycle waits so long that once it is done, the next Brush Cycle should already have been completed. In this case, the universe could travel back in time to produce the next Brush Cycle directly after the delayed one. This has the result of slowing down in-game time, but this happens

for constant cycle length either way if delays get too long. This pattern requires that prediction cycles do not run during the execution of the synchronous part of the Brush Cycle, but only during the asynchronous simulations. Otherwise, the prediction cycles could already have altered entities for points in time that the engine is about to jump back to. The current Telesto demo implementation uses this technique.

9.2. Lisp and LScript

Script languages are generally deployed in modern game engines. They offer efficient descriptions of complex game mechanics, without having to recompile and link the whole application over and over again. Balancing especially involves many adjustment cycles where values slowly close in on their final values. In the gaming industry, scripting languages are also used to offer separation from engine development and game design. This allows the balancing team to do a complete job without having to call back the engine development for recompiles. Another important feature of scripting languages is to feature modding capabilities for players, which might have an important impact on the popularity of a game. It especially increases the long-term player counts, as users will make their own content and deliver new game experience without having to be paid.

However, the way a scripting language is used, differs largely from engine to engine. World of Warcraft (2), for instance, only uses LUA (50) to for interface actions. A LUA runtime is implemented in the local game clients and all interaction with the actual game - and the MMOG server - happens via events and messages. Other examples, like Supreme Commander (21), describe a large portion of their actual game mechanics in the scripting language, like spawning missiles or dealing damage. If this approach is used, only low-level functions are made visible to the scripting language.

Telesto uses its scripting language, LScript, for almost everything; from game mechanics descriptions and GUI up to configuration and resource descriptions. The script engine atoms are even used for shared memory management. This is possible because LScript is very tightly integrated into the game engine. LScript is implemented in the Lisp namespace. This might seem odd, but LScript is actually just the syntax and the parser implementation with some parser features to ease development and a bytecode compiler. The system running behind LScript is a Lisp implementation with optimization and adjustments to fit best for the requirements of a Strategy Game engine. Lisp is an ancient functional language, originally specified in 1958. It features no real syntax - the syntax is a direct description of the abstract syntax tree, but modern Lisp implementations usually feature some "syntax-candy" to make working with a prefix notation a bit more enjoyable. Even with an age of more than 50 years, Lisp is still a very capable and efficient system. Many more recent languages like Smalltalk (60) or Haskell (61) are strongly influenced by Lisp and share the basic structures and ideas. The charm of Lisp is its minimalistic nature. A basic Lisp runtime contains a handful of classes and a 10-lines parser, so it is implemented in about half an hour. Therefore I often use Lisp as a foundation to create my own, project-dependant functional scripting languages. In order to understand the changes made for Telesto's Lisp implementation, we should first summarize the requirements and characteristics of a game engine in respect to its scripting.

Fast Evaluation	A game engine updates hundreds or thousands of Game Objects per cycle. If the scripting language is used for game mechanics descriptions, the evaluation time of an expression becomes critical.
Small Memory Footprint	Besides pure performance in terms of evaluation speed, the memory usage of resources is generally an important magnitude for game development.
Garbage Collection friendly	As .Net is a garbage collected language, it can be of importance for performance optimizations to tune the scripting language in respect to its object creation and destruction behavior.
Fast Parsing	Scripting languages are often used for many different purposes in a game engine. As in Telesto, if the scripts are intended to also store large data, such as models, the parsing speed becomes a relevant factor. A byte code read/write might be required.
Locking	Many Game Objects will be updated at the same time, but it is also probable that multiple scripts are evaluated for the same Game Object at the same time. This might involve dealing with locking; a topic most functional languages don't have to bother with.
Repeating Evaluations	In the typical game scenario, there is a large probability for recurring requests. This could include asking for an object's transformation or the hitpoints of a unit for a GUI display.
Known scripts	While the game is running, all potential scripts in the complete system are known. This is because a user does not insert new scripts, but interacts with the game via lambda calls (such as a button press). In contrast, an application will have to deal with user-created scripts at runtime.

9.2.1. Avoiding Lisp call-chains

The basic and most important element in Lisp and other functional languages is the lambda. A lambda is an anonymous function, allowing functions to be passed around just like normal information (integers for example) in procedural languages. The name is derived from the Lambda Calculus, a generic function description system. Dealing with functional languages is very rewarding. My whole view on problems and the way I write programs changed as I started to venture into Lisp and Smalltalk years ago - and, at least from my perspective, my coding improved. That's especially helpful as functional concepts, like lambdas, are returning to the procedural world at the moment and there are several widely spread functional languages, like LUA, or new ones appearing at the horizon, like the newest .NET child F#. Dealing with the very interesting details and the opportunities of functional programming would go beyond the scope of this document. I will state the most important facts about Lisp as I talk about game-relevant aspects, but if you want to read further, I may recommend Lambda the Ultimate (47), a good site about all kinds of functional languages.

An extra constraint is often added to a lambda: its result only depends on its arguments. In the context of this paper, such a lambda will be called pure-lambda. Common functional languages contain both pure and normal lambdas, with the great majority being normal lambdas. Both pure and normal lambdas contain a list of arguments they expect and a body, which contains the actual procedure and returns a value. The following example shows a lambda which returns the maximum of two numbers. Note that, unlike in procedural functions, there is no "return" statement, as every function in a functional language inherently returns a value.

```
;A lambda expression in Lisp:
(lambda (m n) (if (> m n) m n))
```

```
//The same expression in LScript:
Lambda(m, n, if m > n then m else n)
```

Lisp defines another important element: the environment. An environment is just a list of key-value pairs which contain atoms or lambdas and their corresponding names, which are addressed as variables. A function call is implemented by creating a new environment, containing the passed arguments. The global environment (or the environment in which the function call took place) is then appended at this so-called closure-environment, by adding the global one as a parent to the closure-environment. The body of the function (the lambda's body) is now evaluated on the closure-environment. If a variable is accessed, which is not present in the closure-environment, the request will be passed to the parent-environment. If the lambda's body contains a new lambda call, another closure-environment is created and the existing closure-environment is added as the parent. This behavior can lead to long parent-child chains of environments during evaluation. This approach is simple and powerful, as it allows easy implementations of closures (variables used in a lambda are still available even if the context they were created in is no longer available). Yet, from a performance perspective, this could lead to quite suboptimal behavior. Consider 10 chained function calls (nothing uncommon in a functional runtime), and the 10th lambda requests a global variable, stored in the first root environment. This would require it to climb through the complete chain each time the variable is used. This approach also creates one new object per lambda evaluation, which is - in most cases - immediately disposed of again once the evaluation is done.

In order to improve these issues, a question has to be answered first: do we need closures? Closures are a powerful feature for runtimes. Yet, in a game, a player will not type new lambdas (besides a development console, but that's not the default in-game situation). Instead, all potential lambdas are already present as the game is loaded. New ones might be constructed during the game, but if so, they are the result of other, existing lambdas and therefore expectable. This allows one to relatively easily wrap a closure by saving the required variables in public fields and avoid naming

conflicts, if the closure pattern is required. With discarding closures, call-chains are gone and all lambdas can directly be evaluated on the main environment, without creating and destroying closure-environment and without multiple function calls per variable request.

9.2.2. Caching

During a game, there is a great probability that certain game mechanic elements are static over large periods. This is unlikely in a First Person Shooter, where each Game Object which should be processed is within the sphere of influence and therefore likely to be influenced by a player. An omnipresence-based game like a Strategy Game will have to refresh a lot of objects per cycle, even if they are doing nothing. Consider the function to receive the current hitpoints of a character as an example. The calculation might get rather complex. Temporary effects like being close to a hero unit or increasing the maximum hipoints, as well as damage received might affect the result. Yet, in the vast majority of cases, a vessel is undamaged and temporary buffs are usually used in combat. These buffs are not used during the rest of the game, which implies that these special effects are probably static for large periods of time, too (perhaps until a new armor type is researched). The hitpoints function also has a large probability to be called multiple times per cycle, not only for game mechanics like the update function of a medic to scan if a unit to heal is nearby, but also for GUI operations, such as filling a health bar.

A caching algorithm on a lambda level would be perfectly fitting for this scenario. This is a good example for the special requirements game development requires. Implementing a caching algorithm on lambdas for a generic Lisp runtime would be quite useless, as we have no idea of the potential input parameters. It might go well for some lambdas, but implementing a cache for the "max" lambda in the example above would be a waste of performance. In a game engine, we have two conditions that make a cache table more attractive. First, a game engine calls the same base functions - an Update() function - on Game Objects every cycle. It is therefore likely that a single function is called often. Second, the maximum variety of scripts within the runtime is static as the game is loaded. "Maximum variety of scripts" might sound a bit vague, but think of it as a certain predictability of processing by taking the actual usage of a function into account. The "max" function from above has no meaning, as it is just a function returning the maximum. We can't say much about the potential arguments that it will receive, and therefore we can't really expect the usefulness of a cache. A function that deals damage to a unit in a game has a meaning. Likewise we know a lot about its potential arguments. For example, damage won't be negative. Furthermore, if we take a look at all the other scripts, we could even produce a finite set of values (or value ranges if we take variable damage into account) which could be passed to the respective lambda. Speaking of probabilities, it is much more likely to gain a performance boost from caching a damage function, than from caching a max function. This is a design pattern seen in many good and advanced game engines. Features are not only analyzed by their function, but also by their meaning. This is in order to get the best result for the in-game situation, either by adjusting the implementation, the function, or the meaning for better performance. I often summarize this behavior with the "Content versus Technology" principle.

Let’s go back to the caching now. Would it be possible to implement generic caching on any lambda? Probably not. Just consider a lambda accessing a global variable. The global variable could change and alter the results, which would invalidate the caching result (given the cache is done on the arguments). This means that efficient caching is only possible on pure-lambdas, where global

variables are forbidden and no further lambdas with side-effects are called (consider a hardcoded function that accesses a non-script element, like receiving the transformation of a Game Object). In reality, almost no lambda in an in-game script would match these requirements. If they do, they are so simple that caching would probably be slower than evaluating the result. It should not be forgotten that checking for a cache result and receiving it from the cache table takes time, too.

In order to deploy caching, the chance of a lambda being cacheable should be increased. Reducing the number of side-effect function calls is neither possible, nor recommendable. The scripting can't be disconnected from the non-script elements, as it is part of the in-game universe. If it does not communicate with the rest of the universe, it could be left out entirely. Instead, the scripting language should be integrated into the engine as closely as possible, which means that the side-effect calls should mirror the actual Game Object APIs tightly. This allows one to use the best alternative for a function in the scripting language to receive a maximum of performance. Consider a weapon firing. A missile launcher might check if there is at least a certain distance of free space ahead of it, in order to avoid damaging the unit doing the firing with the missile's explosion. To do so, the missile launcher could do a ray cast. The physics - or a spatial container - generally offers multiple ray casts for different situations. One might only return the first result found and terminate right after, just to check if there is anything intersecting the ray, another one might accept a maximum or minimum distance to greatly narrow down the number of checks required. If the scripting language does not mirror this API, but only offers a generic ray cast that probably accesses the default ray cast implementation - returning all objects intersecting the ray - a lot of potential performance increase by using a better fitting ray cast is lost. A ray cast with a maximum range returning and aborting after a first intersection would be a much better fitting choice.

For LScript, I tried a different approach with the following rule: if a lambda does not call a side-effect function (denoted with a "!" just as in Lisp) it is a pure-lambda. In other words, there are no global variables as everything is passed as an argument. Now this sounds like the complete Armageddon of usability. Having a programmer pass everything required by the lambda (and all lambdas that the lambda might call itself) is of course impracticable. In a commercial project, this would probably lead to more service calls from the scripting teams than giving them access to the source code.

To solve this issue, I introduced a new field to the lambdas. In addition to the lambda-body and its arguments, now a lambda also contains a list of required variables, or just requirements to make it shorter. Once a lambda is evaluated, it asks the environment it is evaluated in for all the arguments, just as a conventional lambda would. In addition, it will also ask the environment for all its requirements. The requirements are a list of all global variables used within the lambda itself and all lambdas it might call. This allows a developer to write lambdas just like in any other functional language and use variables freely, but from a lambda's perspective, its result still only depends on its arguments and requirements and therefore it is a pure-lambda. Caching can now be used on the arguments and requirements. Let's take a look at an example. The following lambda is used in the GUI scripts of a Telesto demo. It is part of the OnClick event handler and checks if the GUI element contains a proper script identification and calls a guiClick lambda. Don't get confused by the "#-1" notation. In LScript, each number is a float. Array accesses are realized via identifiers, which are denoted by a "#" and are similar to integers. The same result could be achieved by using numbers, but differing between numbers and identifiers allows some more performance adjustments for runtime evaluation with an acceptable usability loss. Calling array element #2 is still rather intuitive.

```
1 //Check if a GUI element contains a script ID and
2 //perform pre-event guiClick operations
3 Lambda(id, if not (id = #-1 or id = guiParent)
4             then guiClick(id))
```

So what are the requirements of this lambda. Obviously, it only contains a single argument, "`id`". Once it is constructed, the lambda checks its body for all variable-type atoms and puts them into its requirements list if they are not part of the arguments list. By doing so, the requirements list now contains "`guiParent`". Now, the lambda checks if it calls other lambdas and copies their required lists into their own, discarding duplicate entries. In this example, the requirements list of `guiClick` would also be added.

Once this lambda is evaluated, it asks the environment for all arguments and requirements. From this point forward, no future lambda call resulting from this initial one, will ever have to talk with the environment again. All potential arguments and used variables are already present (except if a lambda wants to set a variable to a new value - I will talk about this case shortly). This bears another great benefit: the evaluation interaction with an environment is reduced to a single "flash" rather than an ongoing "conversation", such as in conventional Lisp. This is particularly beneficial to locking scenarios where multiple lambdas are evaluated on the same environment in different threads.

This pattern to enable caching requires overhead when a lambda is created, as its body has to be traversed in order to construct the requirements set. In the scenario of a game, this is an acceptable overhead, as most - if not all - lambdas are present in the script files and are constructed during load time. If lambdas are created during runtime, this should always happen in reasonable numbers, where the overall gain of performance through caching is greater than the loss of performance due to constructing the requirements list. If a situation should arise where a lot of lambdas are created during runtime, changing the design to reduce the number of dynamically created lambdas should be considered. It's the nature of most performance improvements to make some assumptions on the expected usage scenario. The reasons for my assumptions are based on the analysis of different games. Yet, it is not possible to gain perfect performance in every situation. Although I can't imagine a scenario where this might be required (perhaps a very context sensitive GUI, but on the other hand, the GUI is always rather limited in its GUI object counts and therefore can't lead to too many lambdas created in one cycle), if a usage scenario of a game involves many dynamically created lambdas, caching via a requirements list might be a drawback rather than an improvement.

Let me now continue to the tricky part: !set. For those less familiar with Lisp, this refers to changing the value of a variable. Considering the above example, what happens if the value of `guiClick` is changed? This might result in a new requirements list for it and that would alter the requirements list of the whole lambda. Indeed, if a variable is being changed, all lambdas of the same environment, which include the changed variable in their requirements lists should be refreshed. Likewise, if their requirements lists changed, a recursive refresh kicks off. This sounds like a lot of overhead, but consider the following chart. All lambdas of an environment only have to be scanned and adjusted if the change of the variable actually changed its requirements list. If the requirements list vanishes as the lambda was replaced by an atomic value, it's a design decision whether to update all lambdas or not. As long as variables cannot be deleted (they can't in Lisp), the worst thing that

could happen is that a lambda asks for a few values it won't need. The requirement lists of the new and old values could also be analyzed, to identify if the new requirements list is a subset of the old requirements list, but that's mere optimization.

	New value is atomic	New value is a lambda
Old value was atomic	***No update required***	***Update required***
Old value was a lambda	***Update is optional***	***Update required***

I will supply another assumption here. In the vast majority of cases, variables which change their value often are atoms (floats, strings, vectors, ..). This holds true for many game situations, such as adjusting the damage of a weapon, the hitpoints of a unit and similar values. If more complex mechanics are required to change repeatedly, like the damage characteristics of a spell in an MMOG, the script should be kept static and configured via atomic variables. Let's consider a fireball spell which deals variable damage depending on the type of an affected creature. In this situation, damage dealt is a lambda rather than an atomic value. If these characteristics have to change often - for example by making the fireball more deadly against humans, Orcs or Elves - the actual damage script should be kept static, and a "BonusDamageCreatureType" and "BonusDamageModifier" field should be added. These fields will serve as atomic configuration parameters for the damage script, which will not invoke an iteration over the environment to update requirement lists. Game scripting involves many performance considerations and this would be a good example. Nonetheless, this pattern is rather intuitive and used by many games - like Fleet Operations - even if they do not use this particular caching technique.

Now that we thought about how to deal with the consequences a !set might include, we should think about how it can be used. Following the design principles we introduced so far, a lambda only communicates with the environment at the moment the evaluation begins and all arguments and requirements are received. If so, how can a !set be implemented which is nested in a lambda? A !set requires access to the environment to change the value of a variable, after all. The answer is rather pragmatic: LScript requires a !set to be the first statement. As long as only one !set is contained in a lambda, this works out well. Consider a lambda that somewhere in its body sets a variable a, to 5. This statement can be rewritten, so that the !set is the first statement. Line 1 would not be valid in LScript, but the example is probably easier to understand without switching from Lisp to LScript and vice versa.

```
1  Lambda(cond, if cond then !set a 5) //invalid statement
2  //can be rewritten into
3  !set a Lambda(cond, if cond then 5 else a)
```

There are two drawbacks of this method. The first is a rather uncommon way of formatting these kinds of statements, especially for a developer which is used to procedural programming. A functional programmer won't have many issues with this notation, as that's the way a return-value-driven system works. This issue can be solved via an IDE, which could rearrange these statements in the background automatically. The operations are not very complex, even for nested lambdas. The

second drawback of this method is that a variable might be "changed" to the same value it already has. This produces a small overhead itself, but more important, if the value was a lambda, this might result in an environment iteration to refresh requirement lists. This can be avoided by comparing the requirement lists of the old and new value first, before starting the requirements update iteration.

Still, there might be statements with multiple !set operations nested in a lambda, which cannot be transformed into valid statements in this way. To start with, Lisp usually avoids working with too many global variables, so a clean Lisp code will already avoid this problem. If necessary, the only option is to offer an API call, which will dispatch an asynchronous message to the Game Object (or an equivalent structure which holds the environment, like a GUI object) and add an event handler which performs the !set. This is of course an expensive overhead, but after getting used to functional programming, it is not that difficult to write complex game mechanic scripts, which utilize !set rarely and never use more than one !set per statement. The main issue is to break with procedural thinking and move on to a return-value-oriented point of view.

Now that all these issues are settled, caching can be implemented. The only question remaining is how to actually implement the caching. All arguments and requirements will have to be boiled down into a unique (unique! hashing is not an option for game mechanics. We don't want to kill a player due to an unexpected hashing match) ID, which can then be used to access a value of the cache table. Many policies can be deployed here and the algorithms to generate the ID can be of varying complexity. For LScript I implemented a rather pragmatic, but fast to process solution.

In LScript, only lambdas with less than 5 arguments or requirements are cached. This was a reasonable number for me. The larger the number of arguments or requirements gets, the less likely it is that all of them will be rather static in order to get a benefit from caching. Even if a lambda is rather static, but receives more than 4 variables, it usually calls other lambdas to process the complex task, which may then use caching on their own. This will just shift one level down. If less than 5 variables are passed, their type is determined and a unique ID is created for each variable. The IDs are then ordered and appended to receive the final caching ID. This is only valid if all variables are atomic. As soon as one of them is a lambda, the caching is disabled. This avoids the potentially complex traversing of lambda constructs to produce their ID. It is rather reasonable as well, as a passed lambda is unlikely to be static in a well designed Lisp system, where configuration variables are used.

Last but not least, the caching tables should not be implemented in the lambda itself, but at a global position. To do so, a longer ID is created over the body of a lambda. If this ID creation is robust to reordering of statements without changing the result, then it will allow multiple equal lambdas from different environments to utilize the same caching, which is very useful in a game situation. For example, in a real-time Strategy Game, where many identical tanks move through the field. All their hitpoint functions are the same, but they will all be located in different environments - the environments of their Game Objects.

Finally, to close this chapter about caching, complete caching is optional. If no statement is given at a lambdas declaration, then the parser decides whether to use caching or not. This can follow many policies, from a default value, up to a more complex analysis. Otherwise, the keywords `cached` and `uncached` can be used to force the caching state of a lambda. There are also plans for a conditional

and reactive cache mechanism, but evaluation has to be done first, to determine whether the processing overhead of conditional caching is worth the effort.

```
1  //part of the damage event handler, shortened
2  Cached Lambda(damage, damagetype, mitigation, ... )
```

The above example shows the declaration of a cached lambda with three passed arguments. One requirements variable is used in the body, which is left out here. The function calculates the precise damage dealt by a weapon to a certain target. It is quite likely that this function is called multiple times with the exact same parameters. For instance, every time a unit of the same type is hit by the same weapon. As the damage calculations can become rather complex, this is a rather easy way to construct low-level optimization. It is recommendable to disable caching on certain functions. For example, if they receive a location vector from the physics engine or a similar construct. A physics controlled Game Object will never reach the exact same location in the game world ever again, so caching is a waste of both performance and memory. At the end of the day, if not over-engineered and used wisely, caching can be a powerful performance boost for omnipresence-based game engines. The following figure contains benchmark results for different caching patterns.

	System A, no caching	**System A, default caching as described**	**System A, forced caching on everything**
Idle Scenario	ca. 1.8 Million Cycles/s	ca. 1.8 Million Cycles/s	ca. 1.7 Million Cycles/s
RTS Space Battle	ca. 1.0 Million Cycles/s	ca. 1.4 Million Cycles/s	ca. 820k Cycles/s
MMOG Space Battle	ca. 1.1 Million Cycles/s	ca. 1.3 Million Cycles/s	ca. 650k Cycles/s
Massive "Message Bombardment"	ca. 320k Cycles/s	ca. 860k Cycles/s	ca. 610k Cycles/s

9.2.3. Variable Indexing

The first performance optimization for LScript was the indexing of variables. This was rather implementation-driven. Besides a call to evaluation methods, the most used function calls in the Lisp namespace - according to measurements made in a default test scenario - are calls to receive values from an environment. I therefore chose this function for optimization to start with.

A straightforward implementation of a lisp environment is a .NET dictionary, a generic key-value-store, which will hold a variable's name as the key and its value as the value. While the .NET dictionaries are rather fast (implemented via hashsets), the good old array will always faster. This led to the introduction of LScript header files (interesting how hated C++ constructs reappear once performance optimization begins). Besides some management and meta information, the header files contain a list of variables an environment may hold. This means that an LScript environment - unlike a Lisp environment - cannot hold any variable, but only those which it offers. This is similar to

the public fields an object in an object-oriented language has. This might sound very limiting, but in practice it's no real limitation. Again, a game is not a dynamic console like a generic Lisp runtime. It is a defined system and the scripting language is used to describe the system. A certain environment - for example a Game Object - has a defined list of attributes per se and dynamic or temporary variables are not stored in variables in a functional system, but passed as arguments. Consider the indexing of variables as a tribute to object oriented programming in a functional language.

In the current build, the header files have to be written by hand. The first feature an LScript IDE will probably offer is their auto generation. If the whole game scripting is developed in an IDE, then it is possible to receive the precise list of variables used in an environment without having the user to specify them. This will generate the illusion of conventional Lisp environments which may store any variable. The following example shows a simple header file, described in LScript. The "`@`" statement at the beginning of an LScript file describes the namespace (which header file this environment uses for initialization) of a file. An environment using this exemplary header would start with "`@test`".

```
1  //test object for collision tests 1.2
2  @EnvironmentHeader
3  !set Namespace "test"
4  !set Attributes ["MaxHealth", "CurHealth"]
5  !set Init "testgame\h_testobject_init.lsc"
```

As all environments now know the number of variables they have to provide, their dictionaries can be converted into arrays. To do so, all variables get indexed by the parser. All variable names like "`MaxHealth`" are converted into identifiers by utilizing a global registry module, the Environment-Manager. The EnvironmentManager will map the variables of a header on consecutive numbers. A newly created environment will receive an EnvironmentBlueprint from the EnvironmentManager, which contains the number of variables and an offset. Receiving a variable from the environment now only takes two operations: subtracting the offset from the incoming identifier and performing a direct array access at the resulting position. That helps speeding up the second-most-used operation in the whole Lisp namespace.

On a side note, strings are converted into identifiers, too. They are also stored at the Environment-Manager. This decision was not made due to performance consideration - string operations are pretty fast in .NET - but to establish a central authority for localization. This feature is not added, yet, but it's one of the higher priority tasks for one of the next major versions. Nothing is worse than running through hundreds of script files to search for strings in order to put them into a localization form. That's just what I had to do for Fleet Operations. That's nothing you are eager to do twice in a lifetime.

9.3. Resource Manager

Every game requires content, and the content has to be accessed by the game universe or the extractors. To do so, a game engine utilizes the same mapping an operation system would: it builds up a file system. This is generally achieved by indexing resource files during the load-up phase of the game engine and storing their absolute filepath in a hierarchical tree. A resource's file path - for example to load a texture on the graphics card - can now be received via the path in the hierarchical

tree. This is often represented by either resource IDs, similar to IP addresses, or a resource-manager filepath. The latter is definitely the more intuitive option for human use. Besides this basic functionality, a resource manager is often responsible for processing the resources. This might involve reading script files, loading and converting geometry for various usage scenarios (a physics engine requires geometry in a different format than a renderer), transferring resources to the graphics card and managing their unloading. In many engines, this leads to very large and hard to maintain code blocks in the resource manager, as parsers and loaders tend to result in ugly code. It is absolutely not uncommon to end up with a highly flexible engine with a lot of potential for modularization, but having a bulky resource manager which has to know about every module in the engine. In many cases, the resource manager becomes the true central node of an engine architecture, where everything has to register first. This is neither a flexible nor a maintainable solution. Developing an efficient - efficient in both performance and development - resource manager is one of the most important early achievements a game developer should aim for.

Content tends to get large. Fleet Operations, for instance, consists of over 1 GB of just textures - in compressed format. With that in mind, it becomes reasonable to not load all the resources during the game's load-up period, but load them on the fly as required. This avoids loading unnecessary resources, too. If a player enjoys a round of a real-time Strategy Game which would offer three different factions, but only two are present in this specific game round, all the models and textures of the third are not required. For this reason, all games with rich content load at least their most heavy-weight resources like textures and models during a game. Again, shooters and similar genres might be an exception here, as they have rather simple content requirements. Besides the current map, most resources are always required, like the weapons, player models or GUI components. Loading heavy-weight resources like textures on the fly will result in unacceptable game stuttering if the resources are being loaded synchronously. Dumping a texture onto the graphics card can easily take a second. State of the art games are therefore asynchronous on-demand resource managers.

The Telesto resource manager loads files asynchronously and on-demand, but it is also distributed, meaning that there is not a central authority to oversee the resource loading and unloading process. This is achieved via exchange formats and descriptions. The main resource manager instance running in the Core just creates the file system upon startup and loads all script files. It neither loads nor unloads heavy-weight resources. Instead, these resources are described in short script files (a texture description for example might just contain the filename of the texture, but also extra information like animation) or are completely composed of script (models are stored in LScript, for example). If an in-game entity invokes resource loading, the description is passed to the external module which requires the resource. For example, this could be a new game entity, which tells the renderer to be available for future rendering. This involves loading a resource like the texture. The texture description is passed to the renderer. The renderer now reads the description and requests the absolute file name of the actual texture image from the resource manager. The only thing the resource manager knows about the texture file is its file name. It never read the texture, so it returns the absolute path - the location of the texture on the hard drive - to the renderer. With this absolute path, the renderer now invokes its own resource loading, as every external device features its own resource management instance. It would also be possible to share the same resource management instance between multiple modules. However, that's a rather rare case, usually only happening during development, where multiple renderers, for instance, are connected at the same time. The

following diagram shows an example of the texture loading process, utilizing an external asynchronous extractor, the Renderer, which uses a Renderer Worker to do the asynchronous resource loading. More about extractors in a later chapter.

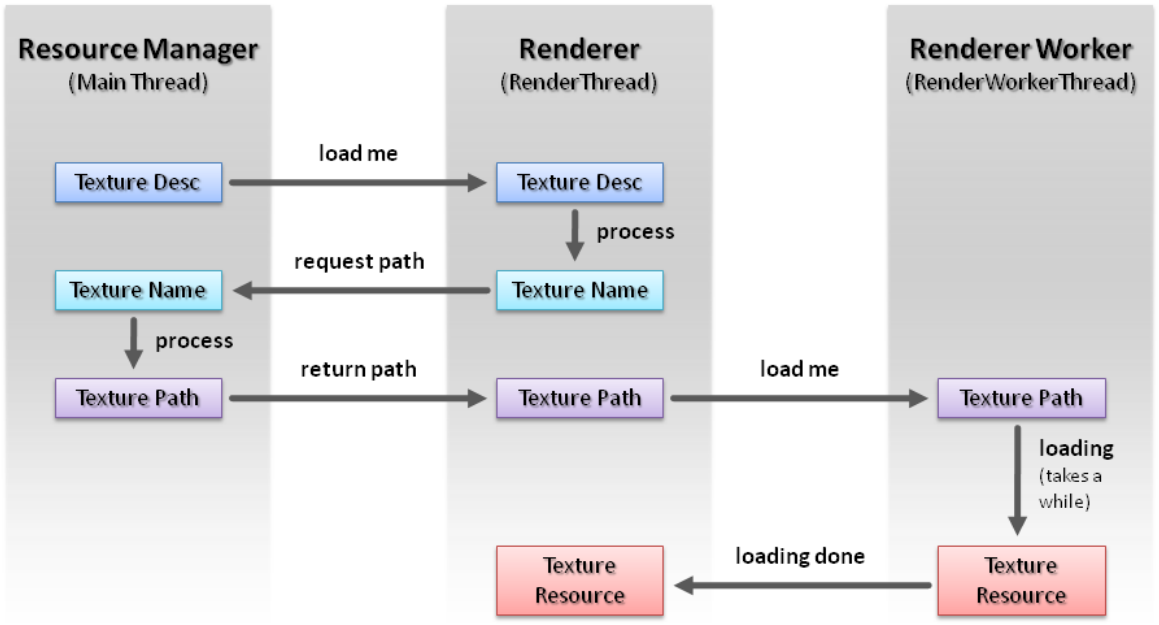


Figure 13: Texture loading

This architecture has some unique points worth talking about. The first obvious change from a common resource manager is the fact that the resources are not handled by a central resource manager node, but by the external modules themselves. This comes with the drawback that every external device (or every group of external devices if multiple externals share one resource pool) has to implement the resource loading itself. In some cases, this will be a development overhead. However, there are some important benefits, too. The external modules now supervise their resources and do not have to follow a generic resource manager API. This allows a lot of performance tuning. The renderer does not need to store its resources in a file-path based hierarchy. A simple collection might be sufficient or just referencing them in the Render Objects. The external devices are also able to optimize the actual loading process precisely to their needs. The renderer might just require a "loading complete" message once the loading is done, while another external device might require more information during the loading process, for example if streaming is involved, like what sound engines do. Another optimization is that not all external modules require asynchronous loading. An AI will probably just work on unit descriptions, and they are plain script files which are already available either way and don't require asynchronous loading. The external modules may now also decide whether or not to unload resources, and if so, under which conditions. The unload-API in many resource managers is the dirtiest block of code of the whole engine, ranging from reference counting through periodic checks. An unload mechanism tailored to the needs of one specific module will be much more efficient, or at least easier to maintain.

The bottom line is that the distributed file manager of Telesto allows one to adapt the functionality of managing resources closely to the needs of the different modules to gain performance or maintainability bonuses, at the price of additional implementations. It also increases the

modularization of the whole engine, as the Resource Manager does not have to know anything about the external modules. It's the job of the external modules to read and understand the descriptions or exchange formats. They serve as an interface an external module will have to implement in order to work with resources. In my opinion that's a much cleaner way to implement resource managing, even if a certain implementation overhead - and perhaps even a small performance overhead to read the descriptions - is involved. It follows the divide-and-conquer principle and allows one to connect or disconnect external devices with a single line of code (this was my mantra for Telesto's modularization). At the end of the day, it's a design decision. There are countless and very popular and successful titles out there, utilizing a central resource management, like Star Trek: Armada II (19). I decided to go for a distributed resource management, as maintainability is king if only working as one person on a large software project. Also, it allows me to implement the resource management features one at a time, while a central resource manager would have to offer a lot more functionality to start with.

The description and exchange formats in Telesto are stored in a universal resource container format. This is just a simple LScript file using the `Resource` namespace, which offers an array of content fields to store data. This might not be the most human readable form, as a user has to know what kind of data is expected in the different content fields for a certain format. Yet an IDE for future development will solve this issue. The following example shows the description of an animated explosion texture which uses UV offsets to jump from frame to frame (all frames of the animated texture are stored in the same file).

```
1 @Resource
2 !set version 1
3 !set format #3
4 !set content0 "textures\explosion01.png"
5 !set content1 ["animation\explosionU.lsc",
6                "animation\explosionV.lsc"]
7 !set content2 [1 / 6, 1 / 6]
```

The first two variables (set in line 2 and 3) are straightforward. The version variable allows specification of the description format, in case the specifications will change in future builds of Telesto. The format field is optional and defines which features the resource offers. In the context of a texture description, format three is a harddrive-stored image file with animation controllers and scaling factors for the U and V coordinates of the mapping. Content0, content1 and content2 are the actual resource description, containing the texture path, the animation controllers and the scaling factors.

Geometry data in Telesto is an example for an exchange format, in contrast to the descriptions above. Exchange formats use the same generic resource container as descriptions, but they do not reference other resource files, like the texture file in this example. Instead, the whole geometry is stored in LScript format, generated by a 3Ds Max (58) plug-in from modeled meshes. Geometry data tends to generate very large text files, easily exceeding a megabyte. Parsing these files with the standard LScript parser would quite literally take forever. To solve this issue, a binary format was introduced with a lightweight compiler and binary stream reader. This feature was later extended

during development to allow saving parts or the complete Lisp runtime system, which could be used for saved games or saving character data on an MMOG server. However, these features are untested and not finally implemented yet.

9.4. Extractors

External modules in Telesto are split into two groups: read-only extractors and externals, which include the flow of information from the external to Telesto. Both read and write operations require different handling. The read-only extractors are integrated via an API in the Core and are maintained by the cycles. For the externals there is no generic pattern. They require a new instance in the architecture, which could be implemented as a Causality Brush. They might also use an existing structure, like the AI, which would use the same input handling as a player connected via LAN. This chapter will take a look at the extractors, which collect information of the in-game universe and do something with them, without actually influencing the in-game universe. The renderer is a good example. It simply presents the current universe state to the user. Many externals which write back information into the game universe, but also require gathering information include extractors, too. The AI, for example, has to gather the positions of its units on the map in a real-time Strategy Game. This aspect is implemented as an extractor.

Telesto offers two APIs for extractors: synchronous and asynchronous extractors. A synchronous extractor is called by the cycle runtime in the synchronous window of a Brush Cycle along with the Causality Brushes. This makes sure that the extractor only accesses objects in one iteration, which are of the same time. An asynchronous extractor, in contrast, is started and stopped by the cycle runtime and extracts information in an ongoing loop, completely independent of the cycles.

Gathering the positions, hitpoints and effects on game entities for the player-level artificial intelligence is a good example of a synchronous extractor. Building the AI world requires resolving some causal relations, which requires that no Game Object has already been updated by the next cycle. Consider a tank, for example, which is currently aiming at a hostile barracks. If the barracks Game Object is already destroyed, the AI will not be able to make sense of it. The AI must therefore use a synchronous read connection with the game universe.

The renderer, on the other hand, just updates its Render Objects with the transformations of their respective Game Object. If a Render Object is linked to a Game Object which no longer exists, it will not update its transformations and will receive a message to destroy the Render Object shortly. The renderer requires no causal dependencies. It doesn't matter whether the tank or the barracks are drawn at the same time. The time differences will be too small for a human player to notice either way. The renderer can therefore use an asynchronous extractor, which completely disconnects the framerate from the cyclesrate. If the renderer slows, the cycles can still continue as usual. If the cycles slow, the renderer can still draw fluent movement due to prediction. Even if the prediction cycles are slow, the renderer will still draw at an unaffected framerate, but there won't be any movement of course.

9.5. Input handling

As a goal of Telesto is to implement a synchrony via reconstruction pattern, it is important to carefully design the input handling. The most advanced reconstruction and synchrony architectures won't help if the input does not arrive in time. The Telesto architecture splits input handling into

three major parts. These are the actual recording of input events, their processing, and the application of the invoked changes. The first two steps are part of the Core, while the changes are drawn via a causality brush as described in a later chapter.

To begin with, the hardware events have to be received. The module responsible for this process is the LocalInput. The LocalInput is, just like the renderer, very implementation dependant. If a game design dictates OpenGL (62), a different implementation of input handling has to be used, compared to a DirectX (52) implementation, where DirectInput is available. As the LocalInput is a write-only device (it does not receive any information from the in-game universe, besides management information like the current in-game time), it is just connected by implementing the ILocalInput interface and does not require a complex message system on its own. Its job is just to receive all hardware device events, gather them in buffers, process them and send them to the next (and most important) module in the input handling chain: the Relay. Before we continue, there is one point of the LocalInput I want to talk about. The LocalInput will, of course, not transfer the plain hardware events. They are first sent over to the GUI, a module located in the Foundation. This is done to prevent transferring unnecessary events over the network, such as clicking somewhere where no action would be triggered. The GUI will respond with a GUIEvent to the LocalInput, which will be stored in its buffer and transferred to the Relay. The GUI component should not be confused with the user interface. The user interface in Telesto is split into two modules, the GUI and the GUI Logic. The GUI is responsible for mapping hardware events like a mouse click or a pressed keyboard button on interface events. This involves a framework to place GUI elements like buttons on the screen or the user-configured key mapping. This two-layer abstraction is very common for GUI development, separating display or input handling from the functionality of a button, as it allows a user to configure controls without having other clients to know about them. Telesto would even allow one to have independent GUI instances for each player, opening support for a customizable GUI-Addon system like in World of Warcraft (2), which is one of the key reasons the game is still so famous. The GUI Logic component, finally, implements the event handlers for the GUIEvents and fires common game events on Game Objects. The GUI Logic could be seen as the API for interface development. More about the user interface in the Foundation chapter.

The Relay is the key node for multiplayer coordination. The Relay receives GUIEvents from LocalInput or ExternalInput instances. The ExternalInput is just a module receiving and buffering messages sent over the network. For every player (human or AI) present in the game, a unique LocalInput or ExternalInput instance is created (note that it would also be possible to have multiple LocalInput instances on the same machine, if split-screen game modes are desired). They are then registered at the Relay and are each assigned to a unique input layer. All LocalInput and ExternalInput instances need to send their batch of GUIEvents, along with the timestamp for the Safe Cycle they are meant for, before the respective Safe Cycle is actually executed. All GUIEvents of a Safe Cycle are therefore embedded in an envelope format, called the GUIEvent Batch. Even if no GUIEvents took place, an empty batch has to be transferred. This is required to assure that all remaining players are still in synchrony if a player drops out of the game - a recommended scenario for an MMOG. Consider three players: A,B and C. Player B now loses connection to player A, but his or her batch of GUIEvents arrived at player C and was executed. Once player B is kicked, player A and C might differ over multiple cycles and reconstruction might be difficult. In Telesto, the clients will notice a delayed or missing player immediately once a Safe Cycle is missed. There is also an option to

have all clients acknowledge that they are ready to proceed with a Safe Cycle before applying it. If this option is chosen, the Safe Cycle window is narrowed down a bit, as additional communication overhead is required. The following diagram shows an overview of a generic input handling in Telesto. All actions after the Relay are only executed in the synchronized window in a Safe Cycle.

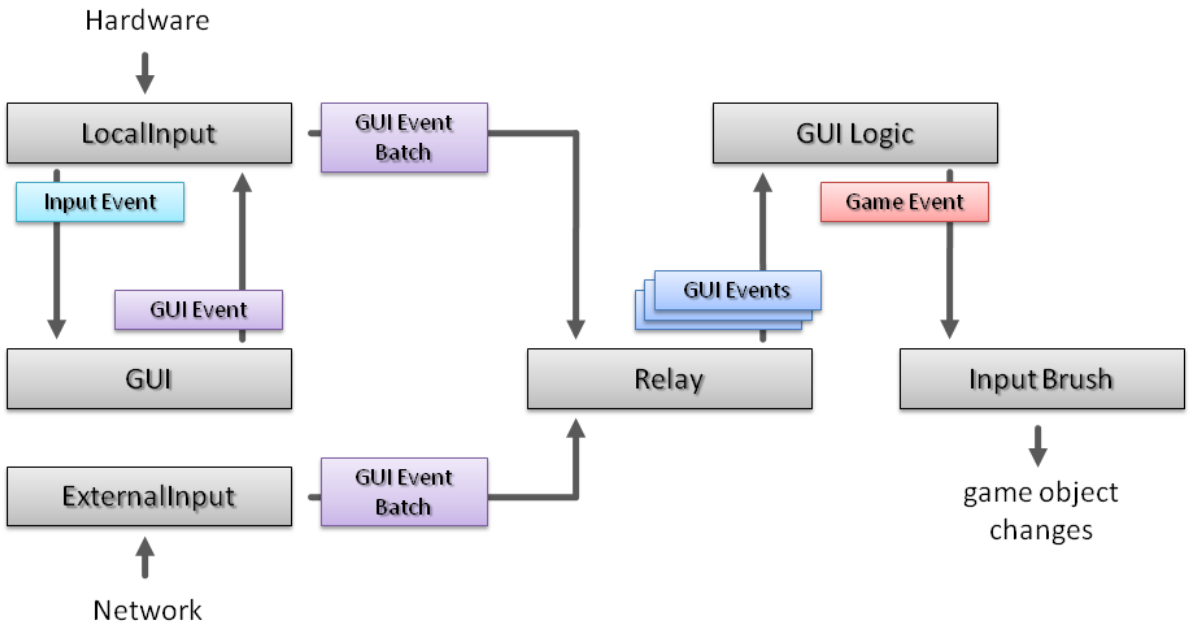


Figure 14: Input Handling

Obviously, there is no server mentioned in the description above. In fact, Telesto utilizes a peer-to-peer multiplayer mode. Only some management functions will require a unique entity in the in-game multiverse, the random generator for example. To handle these issues, one of the clients is declared to be a host (could just be the player who started the game round for example). Besides the default input handling above, the host also does some management operations, like assigning the AIs to the fastest clients, kicking clients out of the game, or distributing random tables. The host can be changed at any time. For an MMOG, it might sound strange to use a peer-to-peer system. Yet, there are several interesting models simulating a vast game world without utilizing highly expensive server hardware. A good example for these techniques is Guild Wars (13), which actually fakes an MMOG world by moving the lobby into the game in the form of cities and outposts. Going into detail about this would exceed the scope of this paper. At the bottom line, it is possible to implement an MMOG based on peer-to-peer multiplayer.

9.6. Containers

The structure that game entities (or their components in Telesto) are stored in has a great impact on performance. The Core holds two important collections of entity components: the GameObjectContainer and the PhysicsObjectContainer. As mentioned in part 1 of this document, it is a recommended technique to implement containers fitting the actual structure of the content to optimize queries. For a Strategy Game or an MMOG, this would be a spatial container for Physics Objects and a logical container for Game Objects, to speed up the most common queries. In addition to this generic purpose, these two containers also serve as a hub to start the respective simulation

and update processes. They are therefore also connected to the Thread Pool to dispatch the physics simulation and Game Object update jobs.

10. Foundation

The Telesto Foundation is the layer of Telesto which is changed most during game development, as it contains all the game-relevant modules to implement the fundamentals of the gameplay, physics and usability. Most precise implementations of the modules contained in this layer are very dependent on the chosen genre. This chapter will cover the most important modules contained in the Foundation and give some examples for a real-time Strategy Game or MMOG. The following figure shows the three basic sectors the Foundation consists of and their most remarkable functionality. Going into detail for all functions of the respective modules would exceed the scope of this document, as a lot of management and optimizations is involved in each of them. I will go into more detail where appropriate without messing too much with an actual implementation.

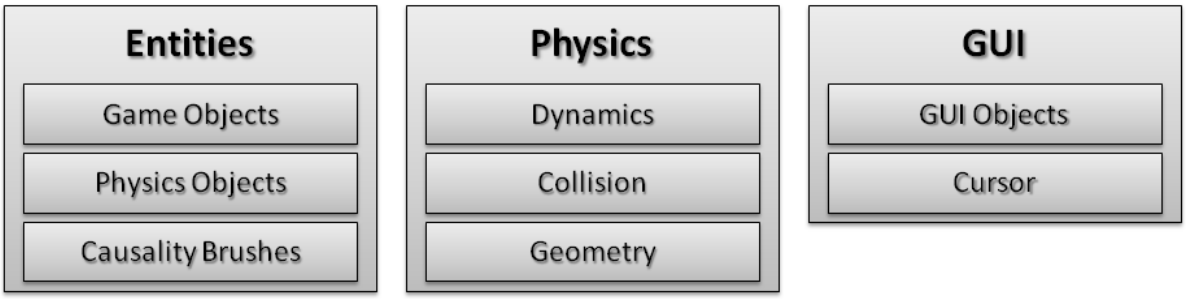


Figure 15: Telesto Foundation Overview (Functionality)

10.1. Entities

One of the most defining decisions during the development of game engine architecture is definitely the decision on how to store and manage the information which represents in-game entities. This structure is more commonly known as Game Objects, like we talked about in part 1 of this document. Telesto uses a shared-memory Game Object model by utilizing LScript environments and their atoms as shared-memory blocks. In order to achieve potential for parallelization, Telesto constructs in-game entities from several components. This allows one to use parallelization-per-function, as all components are designed to be processed without needing any information from other functional domains. The core component is the Game Object itself, which holds an environment responsible for processing the game mechanics. A Game Object might also contain a Physics Object, which serves as an entity for the physics simulation. The Render Objects contain information on how to draw the entities, but they are not stored at the Game Objects, but are kept in the renderer. The following chapter will deal with the special issues and opportunities of Telesto's Game Object model.

The Game Object acts as a central representation for an universe entity. It stores an ID-reference to a Physics Object - if present - but also holds its own spatial state. This is because there can be spatial objects which are not simulated by the physics engine. For example, consider floating GUI elements, such as arrows or special effects like explosions. The Game Object also manages message receipt and their event handlers. Besides some basic management functions, all relevant API calls are wrapped in messages, to assure a completely asynchronous communication with the Game Objects. This is important, as the messages are not executed upon arrival. All state changes are controlled

and executed by the so-called Causality Brushes, in order to assure multiplayer synchrony. Causality Brushes exploit the functional independency of the several Game Object components to perform a high performance causal analysis, or at least causal ordering.

The picture behind the Causality Brushes is essentially that of faking a painting. A pure color analysis on the picture would just lead to a shallow copy. That is because only the result of a long process chain is observed and important side effects are lost - like the subsurface scattering from multiple layers of color, which brings the real "life" to a piece of art. A good fake includes taking a close look at the technique the picture was painted with and drawing all colors on a certain region in the same order as in the original. That's the idea behind Causality Brushes, too. They don't try to transfer a certain universe state, but they recreate the complete process where the final state evolved from and control this recreation in order to achieve synchrony. A Causality Brush performs actions of a specific "functional domain" - for example the spawning of objects or their destruction. To do so, it receives job tasks while game mechanics or physics simulations are being processed. In the example of spawning an object, this could just be a constructor lambda. Causality Brushes are then executed in a defined order. The order does not have to be static, as it can be influenced by Causality Brushes, but the order of execution must be the same on all clients of the in-game multiverse. In other words, as long as all Causality Brushes are executed in the same order as when they received the same, unsorted set of messages, all clients will produce the same final universe state.

As a side note, the Causality Brush pattern is developed from a relatively common architectural principle: the cycle report. This pattern is often utilized in engines of smaller scale. All Game Objects are executed in parallel, but changes are not directly written back on the Game Objects. Instead they are written in a report of changes, the cycle report. Once a cycle is complete, the cycle report can directly be transferred to or compared with other clients, in order to establish a simple synchrony pattern. The cycle report approach is popular for its simple implementation and its capability for multi-threading and synchrony. For a commercial-scale game engine this approach is not suited due to performance issues and limited synchrony optimization for complex causal chains.

We will soon take a closer look at a Causality Brush example. However, before this I want to answer some important questions about them. During development, the question arises about which functions can be processed on a Game Object directly and which have to be applied via a Causality Brush. Generally, all functions that involve a causal chain should be extracted into a Causality Brush. As a rule of thumb, all actions that require access to an external entity should be moved away from direct processing. As long as an entity only changes its internal state, no causal processing has to be done, as it is independent from other entities either way and can be processed at any time and in parallel. Only if actions require information from another entity - or invoke events on entities - they are part of causal chains and bear the potential of asynchrony if processed in changing orders. As this was rather abstract, let me visualize the difference using an example. Consider the native regeneration rate of a starship's shields. They always recharge at a constant rate. All starship Game Objects may increase their shield strength in parallel, as it is unaffected by any other Game Object. This process does not have to be extracted as a Causality Brush. However, subtracting shield hit points always has a causal dependency, for example, being hit by a laser beam. This process is the result of some external call, like one Game Object telling another one to receive shield damage. This process can no longer be executed at any time, because - given the shields are at their maximum - it

will end at a different resulting shield power if the damage was dealt before shield regeneration was accounted for or afterwards.

Now that a set of potential functions is identified that should be handled by Causality Brushes, it has to be determined how many Causality Brushes are required and what each brush should process. To do so, it is important to precisely define the requirements a Causality Brush has in regard to the functional domain it covers. The following table summarizes these requirements.

Causal Independency	The result a Causality Brush processes on each queued entity has to be independent from the time of execution. This means that the actions a single Causality Brush performs should be completely parallelizable. This constraint is required to assure that a Causality Brush is actually the lowest level of a causal atom. It would of course be possible to execute causal events within a single Causality Brush, but how should the Causality Brush resolve these chains? It would have to use Causal Subbrushes itself or other mechanisms. That's not an efficient solution.
Unordered Queue	The order in which entities are queued for processing at the Causality Brush has to have no influence on the result. This is an obvious requirement, as different clients won't perform their initial game mechanic updates in the same order (for example if the machines offer a different number of hardware threads).
No "unbrushed" side effects	The processing of a Causality Brush might produce effects on entities which are not within the input queue. If so, the only way a Causality Brush may execute these side effects is to queue the entities at another Causality Brush. This is important to assure that no hidden side effects are produced by using an external entity, as some kind of storage container to transfer causality outside of the Causality Brush system. This could violate the causal independency requirement. Telesto allows one to queue entities at Causality Brushes which were already executed in the current cycle. Cross-cycle causality blurring is allowed.

A simple example for a Causality Brush is the SpawnBrush. Whenever a new Game Object has to be created, a message at the SpawnBrush is dispatched with the Game Object description LScript file path as a reference, as well as optional spatial information or arguments to handle, over to the LScript environment constructor. Once the SpawnBrush is executed, it iterates over its complete queue and spawns the new Game Objects according to the description file, which might also cause the spawning of Physics Objects or Render Objects. The SpawnBrush also features a callback mechanism, which allows echoing the IDs of created entities back to the environment which requested the spawning of the Game Object. The SpawnBrush fulfills all the requirements listed above. The spawning of an entity is completely independent from the spawning of other Game Objects. If two spatial objects should spawn at the same location, the physics will detect the collision during its next simulation phase and might queue a collision at the PhysicsBrush, but the SpawnBrush itself does not care about that. This fulfills the requirement of causal independency. With that in mind, the SpawnBrush does not care about the order in which constructors are queued. The SpawnBrush is executed during the synchronous window within a Brush Cycle, when neither

physics nor game mechanics simulations are running. There is thus no difference as to which Game Object has been spawned first. Herein the unordered queued requirement is fulfilled. Finally, a callback will affect an existing Game Object. Yet, the SpawnBrush does not alter the environment directly, but sends it a Callback message, which is handled by a separate Causality Brush. The side effects requirement is correctly fulfilled, too.

The SpawnBrush is a good, but rather simple example. A creative reader should easily be able to think of much trickier situations, especially for game mechanics. A first step to a fast and efficient solution is creating a MessageBrush which will handle all messages sent between Game Objects. At the end of the game mechanics simulation, a large queue of messages will have been gathered. The messages should now be passed on to the Thread Pool for processing, which will probably result in new messages gathering at the MessageBrush. The process is continued until either the queue is empty or a maximum of iterations is reached. This very basic MessageBrush already does a good job and could be deployed for a commercial game engine. In order to optimize the usage of the Thread Pool and to utilize causal ordering, Telesto filters the incoming messages into several categories and processes them using different brushes. Examples are the CallbackBrush or the DamageBrush. This can be done by just filtering on specific message IDs. This allows grouping messages of similar expected processing time (like the DamageBrush does) to always have a job at hand for an idling Thread Pool worker.

The Causality Brushes have proven to be a powerful asset of Telesto. They offer good performance and the synchrony-via-reconstruction method offers good network bandwidth usage. Similar concepts are already utilized in game engines. For example, in the simpler form of the cycle report as mentioned above, or in slightly different forms for behavior patterns or as mergers, where central nodes are used to handle causal conflicts, such as having the player want to move through a wall and the physics denying said action. These causal conflicts are solved in Telesto automatically through the order of execution of the Causality Brushes. If the PhysicsBrush is executed last, the physics will always have priority over player actions. In more complex scenarios, a Causality Brush may also send a management action to another brush, requesting to drop an item from its queue. The Causality Brushes build up the functionality required for a synchrony-via-reconstruction multiplayer pattern. If they cause an asynchronous final state due to a wrong implementation, the whole multiplayer mode will fall apart. Yet, due to the very modular concept of implementing different Causality Brushes for different "functional domains", they are quite efficient to debug and this separation keeps the single brushes rather simple and maintainable. Yet again, they are a tribute to developing a whole engine as a single person. However this will always be an efficient concept in a commercial product too, as multiple developers can be responsible for separate brushes, which reduces the communication overhead that delays so many software projects.

10.1.1. Multiplayer synchrony-via-reconstruction

The choice for a reconstruction multiplayer was challenging and changed a lot of the initial architecture plan, but in the end it was worth it. There is very little literature on multiplayer, basically because most games do well with just transferring state changes. Only Strategy Games and MMOGs, which both suffer from large Game Object counts and an omnipresence-based gameplay (from the view of an MMOG server, the game is actually omnipresence-based, as there are countless players active, similar to the units of an real-time Strategy Game). Therefore a commercial example for reconstructing multiplayer is StarCraft (5), which is known to even be playable on 56k modems.

The following table shows some testing results of a Telesto demo running on a network pool of eight machines, a common player count for real-time Strategy Games. The TCP/IP implementation used in this demo was neither optimized nor perfect, so even more bandwidth optimization is possible. Yet the upload and download rates are already rather satisfying. The demo scene used was a generic game situation with 250 Game Objects active. The GUI was ignored as the Test scenario generated direct GUIEvents. APM stands for actions per minute, a common ratio to measure activity (or "skill") in the real-time-strategy scene. A "pro-gamer's" APM is often around 150 with peaks of about 350, while South-Korean StarCraft players range around 400 APM on a regular basis with peaks over 500. The kilobits were measured at the Relay, counting the size of TCP/IP packages. Some additional networking overhead might occur, but this should not change the result significantly. The numbers are not final, but can be seen as expectations of what a synchrony-via-reconstruction engine might offer.

	Average Upload per Client	**Average Download per Client**
8 Clients, all idling: 0 APM	ca. 1.10 kb/s	ca. 1.12 kb/s
8 Clients, 100 APM	ca. 1.42 kb/s	ca. 1.76 kb/s
8 Clients, 250 APM	ca. 1.98 kb/s	ca. 2.04 kb/s
8 Clients, 500 APM	ca. 3.22 kb/s	ca. 3.38 kb/s

The current plans for the MMOG based on the Telesto architecture will also increase the average upload slightly, as it requires having an observer-client in the game, which receives information from all players in the game and sends important information like gaining experience or looting items to the server. In the above chart are some numbers to compare the results. Numbers without source reference are taken from the official documentation, the user manual or my own tests. Precise results depend - in most architectures - on the chosen engine or game genre. This was especially visible in Aion (12), where the download rate greatly varies from region to region, depending on the player density. The speeds for Counter-Strike represent a Counter-Strike dedicated server with 16 player slots used. The built-in voice-over IP of World of Warcraft was not taken into account and was disabled. Diablo II was also measured with the maximum of eight players present.

	Average Upload per Client	**Average Download per Client**
World of Warcraft **(2)**	5-15 kb/s (63)	5-15 kb/s (63)
Diablo II **(64)**	10 kb/s	10 kb/s
Aion **(12)**	8 kb/s	20-100 kb/s
Counter Strike **(65)**	4,5 kb/s (66)	4,5 kb/s (66)

Besides bandwidth usage, the latency is an important point to look at for multiplayer implementations. Yet, the latency of Telesto is fixed by defining the length of the Safe Cycles. The above example was tested in a LAN and the Safe Cycles distance was 50ms, which leads to a maximum latency of clients under 50 ms, as otherwise the engine would start waiting for them.

Online games might of course use longer Safe Cycle distances. The next build of Telesto will also include functionality to allow different Safe Cycle resolutions per connected player. With a Safe Cycle distance of 50ms, a player with a 30ms could arrive on time on every cycle, while a player with a 150 ms ping will only be expected to arrive at every fourth cycle - which will slow down the games reaction on his or her input, but that's the price a slow ping demands.

10.2. Physics

For a long time, the graphics were the most important sales-factor for games, besides the actual gameplay of course. In the last years, the graphics quality and renderer content complexity reached such a high level that a further increase in quality was no longer perceptible enough to act as a unique selling point. That's when suddenly a lot of research was put into physics engines and many of the modern stand-alone physics solutions were born. As already mentioned in part 1 of this document, a physics engine consists of four important features: the dynamics simulation, the collision detection, the collision analysis and the collision response.

Usually, one will not attempt to implement a physics engine on its own. They are probably - besides sound engines - the most common third party software in modern game engines. A physics engine involves a lot of low-level optimization, subtle mathematical models and algorithms and a fast runtime. It would take a well trained and experienced development team to challenge the feature list and performance of the current physics engines, which all required several years of development to get so far. Time no commercial company would like to invest. For Telesto, my goal was to implement an engine for space-based real-time Strategy Games and MMOGs. This setting - space - bears a few characteristics for a physics engine that encouraged me to implement one from scratch. For a space simulation, the probability of collision is much lower compared to a planetary game, where there are always countless collisions just from objects lying around. This reduces the pressure on performance, especially for the very difficult "rest problem" where two Physics Objects lie on each other. The space setting can also be solved quite well with a pure rigid-body simulation, as neither soft-bodies nor cloth simulations or fluid simulations are required.

Still, the physics engine of Telesto is one of its most complex modules. I decided to implement a Newtonian rigid-body simulation. The benefit of creating the physics engine from scratch is that it was possible to closely tailor it into the rest of the Telesto architecture. All third party physics engines (I took a look at PhysX (15), Havoc (41) and ODE (42)) did not support my API and would have required at least one additional wrapper layer to run their own simulation either way and are only accessible via query statements. The Telesto physics engine is now directly able to deploy it's simulations on the shared Thread Pool. This chapter will cover the concepts and the current implementation of the Telesto physics engine.

10.2.1. Dynamics

The first thing to start with when developing a physics engine is the movement of objects. Telesto utilizes forces and torques to express the movement characteristics of an object, so it has to deal with the dynamics of entities. The mathematical background for dynamics is well-known. Many physical models, for example bodies falling down under the influence of gravity, are commonly described as analytic models, like the following example.

$$y(t) = \frac{1}{2}gt^2 + v_0 t + y_0$$

Analytic models are very handy solutions, as they allow one to get the state of an object at any valid point of time by just inserting the requested time t and supplying the start state v_0 and y_0, as well as certain constants describing the system, like g in this well-known example. However, a game engine can't hope to find an analytic solution for their state transformations, as real random is involved - the user input. A physics engine therefore utilizes numerical models, like the following.

$$y(t + \Delta t) = F(y(t), \dot{y}(t), \ddot{y}(t), \dots)$$

These models calculate a future state as a function of the current state and its time derivatives. Intuitively, this is just what a run-time does. It iterates over objects in certain intervals and updates the states based on the old state and the elapsed time, like a multiplying velocity with the elapsed time and adding the result to the current location to receive the next location. This mathematical background is of course true for game mechanics, too. Yet, a developer usually does not perceive lambdas or blocks of a scripting language as a scientific model, while a physics engine is of course very closely linked to the mathematical ideas behind the systems, like Newtonian Mechanics in this case.

So what are the requirements the game engine has on its physics - or more precisely dynamics - simulation? At the end of the day, besides obvious performance requirements, the game engine expects the physics to update the transformation of a Game Object. In other word, the results the dynamics simulation has to produce are the translation and the orientation. The goal of the whole dynamics simulation is therefore to calculate a translation vector and an orientation matrix - or a more compact representation for orientation, like a quaternion. However, that it is an optimization to save memory does not change the mathematics and ideas behind it.

In the years before physics engines were available, there were still moving objects in games. These include floating platforms in Jump'n'Runs or missiles fired in the first shooters. Their movement was expressed by translational and angular velocity vectors, which were just added to the current location and orientation, after being scaled by the elapsed time. This very basic pattern to express movement will of course not lead to a sufficient physics simulation. However, it is very fast to process, and therefore that's exactly what a prediction cycle does. In order to achieve movement without having to process a complete dynamics simulation, the resulting angular and translational velocities are stored and utilized by the prediction cycles to approximate the movement of an object between the Brush Cycles. The following diagram illustrates this process for translation, which is easier to visualize. Orientation behaves exactly the same, just the mathematics behind it are slightly different.

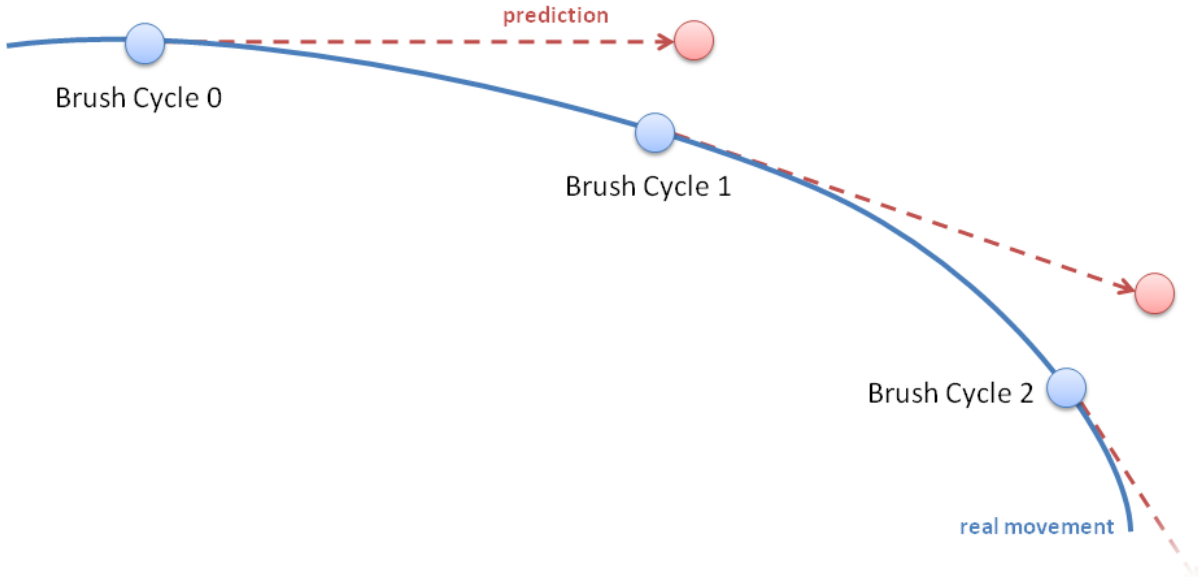


Figure 16: Prediction

The figure above shows a very zoomed-in situation on a very fast movement, to visualize the way a physics prediction works. Slower movement will feature a higher Brush Cycle density and the difference between the predicted next transformation and the actual next transformation becomes much less significant. Yet, it is important to keep in mind that a prediction is just an approximation based on static velocity of the last calculated state. So, this leaves us with the four important values to produce in the dynamics simulation: The translation, the orientation, the translational velocity and the angular velocity. The last things we need in order to set up the dynamics simulation are the input variables. These are the forces and torques applied to an object. Everyone should recall Newton's famous second law of motion.

$$F = \frac{dp}{dt} = \frac{d(mv)}{dt} = ma$$

This leads us to the final input variable that the system needs, the mass m, as well as the intermediate variable which will be calculated in each Brush Cycle, the acceleration a. The linear momentum p does not appear in a dynamic simulation implementation, but is important for collision response. Finally, forces can be applied at any point of an object, as an object can be seen as a sum of points with individual masses, momentums and forces applying to them. Consider a fluid simulation for example: the center of mass is introduced to simplify the dynamic analysis for rigid bodies. This allows the simulation to treat all forces of an object as if they were acting on the center of mass and by doing so, ignoring the point of their application altogether. This assumption, as well as many others, is used in physics engines to produce a believable output at reasonable performance. Unlike scientific simulations, a game does not care for correctness. The only two things relevant are performance and "that it looks good". In this paper, I will not cover the complete mathematical background of Newtonian Mechanics and rigid bodies, but will follow a very implementation-driven approach, presenting the formulae as they would have to be implemented, with the optimizations used by today's physics engines. Simplification and abstraction are key, not only to achieve believable physics in reasonable time, but to simulate the whole in-game universe altogether. For a more mathematical approach, which describes the basics of a physics engine

without optimizations and implementation notes, I can recommend the excellent paper series of Chris Hecker (67). He describes an excellent 2D physics simulation with all mathematical background information and also offers the ideas on how to upgrade it to 3D. A perfect base lecture for writing a physics engine.

for further reading...

Physics, The Next Frontier
http://chrishecker.com/

However, let's get back on topic. With the knowledge about forces and the simplification of the center of mass, it is already possible to put all calculations together to do the processing which happens to the translation of an object during a Brush Cycle.

Input: **The Forces**	Numerous forces might apply to an object, ranging from the engines of a starship to the gravity well of a planet. Forces describe an influence which causes the body to undergo an acceleration. $$F_{Total} = \sum_i F_i$$ Many physics engines differ between global forces, like the force caused by a shockwave or the gravity well of a planet, and local forces, which are transformed by the same transformation as the rigid body itself, such as the force caused by the starship's engines. Whether this is seen as a property of a force or whether all forces are considered to be already correctly transformed into world space before the physics simulation starts is a design decision. Telesto uses the local force concept. Obviously, the summed forces are only valid for the current Brush Cycle.
Prediction: **Linear Velocity**	The acceleration is an intermediate result achieved by summing up all forces and dividing it by the mass of the object. With the simplified assumptions of a physics engine in mind, this is quite obvious. $$a = \frac{F_{Total}}{m}$$ The acceleration describes the change in velocity over time. The acceleration is not stored in the Physics Objects. Like the summed forces, the acceleration is only valid in the Brush Cycle it was calculated. The velocity is the first Physics Object state which has to be stored for prediction. $$v(t + \Delta t) = v(t) + a\,\Delta t = v(t) + \frac{F_{Total}}{m}\Delta t$$
Output: **Position**	The final result required to build the transformation matrix of the Physics Object is the position r of the rigid body. $$r(t + \Delta t) = r(t) + v(t + \Delta t)\,\Delta t + \frac{1}{2}a^2$$ $$= r(t) + v(t + \Delta t)\,\Delta t + \frac{1}{2}\left(\frac{F_{Total}}{m}\right)^2$$

Besides rotation, the above formulae simulate a realistic movement, but they are only valid in a no-atmosphere environment, like space. There are various ways to simulate friction in an atmosphere. Some physics engine map friction as an auto generated force following constraints. However, this could lead to unexpected behavior, especially for very slow movement, as forces processed in a game engine are not completely precise, but just float vectors with their known inaccuracy. This could lead to an object almost reaching a complete stop to actually accelerate in the opposite direction or jumping back and forth. An alternative way is to execute a formula directly on the

velocity, like dampening it by a certain percentage each cycle. Telesto supports both patterns and the current demo uses a simple linear dampening of velocity to simulate the Star Trek (48) movement behavior of vessels in space (which is actually a plane in an atmosphere, rather than a real spaceship).

The angular dynamics are slightly more complex. The angular equivalent to force is the torque. The torque τ can be seen as the influence a force F applied on a point r has on the rotation of a rigid body. Obviously, the point of application matters for calculating the torque. Just pull your keyboard on the upper left corner or pull it closer to the center of its mass and you will see different rotational behaviors. Mathematically, torque is expressed as follows.

$$\tau = r \times F$$

Just as for translation, the angular quantities are measured at the center of mass, instead of all points of a body. For the angular formulae I will not use the angular velocity to start with, but the angular momentum at the center of mass. This can be received from the torques directly, as the torque is - just like for the linear situation above - the time derivative of the angular momentum L.

$$\tau = \frac{dL}{dt} = r \times F$$

Now the question remains as to how to receive the orientation from the angular momentum for both prediction and Brush Cycle processing. The first thing required for a believable rotation behavior is a description of the way an object rotates, very similar to how the mass of an object describes the way it moves for linear dynamics. Consider driving an empty car and a car with its trunk filled with something heavy, like a weekly supply of Cola. If you buy enough, the behavior of the turning car will change. This is described by the inertia tensor matrix I. That's why I avoid working with angular momentum until the very last moment. This avoids messing around with the inertia tensor early on, as it is a matrix compared to a scalar for linear dynamics, which tends to get a little nasty for both my brain and float accuracy. There are algorithms to calculate the inertia tensor for geometry, but most games don't bother to describe the mass distribution over the volume of their models or don't even have closed meshes, where a volume could be calculated automatically. Therefore most engines just use a generic inertia tensor generated over the bounding box of an object. A bounding box with the width w, the height h, depth d and the uniformly distributed mass m has the following inertia tensor I.

$$I = \begin{bmatrix} \frac{1}{12}m(h^2 + d^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2 + d^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2 + h^2) \end{bmatrix}$$

Similar matrixes can be found for spheres or other primitives and there is a vast collection of prefabricated inertia tensors on the web. If a non-uniformly distributed mass over the volume of the rigid body should be necessary - for example for physics puzzles - mass information can be stored in the models to calculate an inertia tensor during the load phase or in the editor. As Telesto is designed for games using a space setting, the generic bounding-box inertia tensor was sufficient.

Obviously, the goal of the whole dynamics simulation is to receive a world-space orientation. This was no problem at all for the linear dynamics, as mass is the same in object space and world space. The inertia tensor, however, changes as we move between object and world space. A physics engine therefore stores the inertia tensor in object space (I_{Object}), to avoid calculating it in every cycle. A simple bounding box inertia tensor might not be that much of a problem, but a good physics engine architecture should not deny the possibility of using more complex inertia tensors which cannot be calculated efficiently at runtime. The world space inertia tensor (I_{World}) is then achieved by performing a similarity transformation with the orientation A. As a rotation matrix like the orientation is orthogonal, its inverse is its transpose.

$$I_{World} = AI_{Object}A^T$$

Now the only thing remaining is to connect the inertia tensor and the angular momentum to receive an orientation matrix. To do so, the change of orientation over time is required: better known as the angular velocity. Thus it will finally be necessary to do the jump from angular momentum to angular velocity. Taking a look at the linear velocity and linear momentum, both are linked via mass. Dividing the linear momentum through the mass of the object yields the linear velocity. That's exactly what the angular quantities have to be handled as, too, just with the difference that the angular momentum has to be transformed by the inverse inertia tensor. To avoid calculating the inverse inertia tensor in every cycle, it is common use to actually store the inverse inertia tensor in object space (I_{Object}^{-1}) in every Physics Object. As we are talking about rigid bodies, the inverse inertia tensor in object space will be constant. The angular velocity (ω - don't ask me why it has this symbol) may then be received as follows.

$$\omega = I_{World}^{-1}L$$

Obviously, the angular velocity is a vector, while the orientation is a matrix. We can't just stick them together like we did with the velocity and the current position. Allow me to take a closer look at the angular velocity. The angular velocity is actually a pseudovector, with its direction defining the axis of rotation and its length defining the speed of rotation. A vector is rotated by calculating the cross product between the angular velocity and the given vector. Applying angular velocity to the orientation matrix would require a cross product between a vector and a matrix, something usually undefined. To solve this issue, we will just represent the cross product in its less common matrix notation.

$$x \times y = \boxed{x}\, y = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

Applying the "cross product matrix" of a vector on a matrix results in a matrix where the columns represent the cross products of the vector and the columns of the initial matrix. That's exactly what is required to numerically integrate the orientation to the next Brush Cycle, as the columns of an orientation matrix can be seen as the unit vectors of the target coordinate system. Consider rotating an object around its X-axis. This would require you to both rotate the y and z axis around the x axis. That's what the "cross product matrix" will do, but it is just a more mathematical description. Just to spare myself some writing, I have made up the box notation for the "cross product matrix" of a

vector. Thus that's finally the last element required to set up the angular processing formulae for a Brush Cycle.

Input: **The Torques**	The torques are received from the forces and the same rules apply in respect to local and global forces. Besides that, the calculation is trivial. r is the point of application of a given force. As with forces, the torque is only valid for the moment (the Brush Cycle) it was calculated for. $$\tau_{Total} = \sum_i r_{F_i} \times F_i$$
Prediction: **Angular Velocity**	The first thing required to get the angular velocity is the angular momentum, which is numerically calculated from the summed torque. $$L(t + \Delta t) = L(t) + \tau_{Total} \Delta t$$ To receive the angular velocity, the inverse inertia tensor, which is stored in object space in the Physics Object, has to be transformed into world space. This is just an intermediate result and obviously dependant on time, as the orientation used is the orientation matrix of the last Brush Cycle. $$I^{-1}_{World}(t) = A(t)\, I^{-1}_{Object}\, A(t)^T$$ With the current inverse inertia tensor in world space at hand, it is possible to receive the angular velocity from the angular momentum. $$\omega(t + \Delta t) = I^{-1}_{World}(t)\, L(t + \Delta t)$$ The angular velocity is, just like the linear velocity, the variable to be stored for prediction cycles. Just as with linear velocity, a friction formula might be applied it.
Output: **Orientation**	Finally, we will have to apply the angular velocity to the orientation to receive the orientation of the current Brush Cycle. As described above, we will use the magical "cross product matrix", represented by the boxed angular velocity. $$A(t + \Delta t) = A(t) + \Delta t \boxed{\omega(t + \Delta t)} A(t)$$

With the orientation, we gathered everything required for the dynamics simulation. We should now have covered the math directly involved in the implementation, but there is a lot of beautiful physics around it, which is required if you want to do more complex stuff, such as a fluid simulation or soft body simulation. It is definitely worth learning more about that if your plan is to implement a physics engine, but recreating Newton's work would be way out of the scope of this document. I would rather like to take a closer look at some implementation details.

While the formulae to get the orientation work out on paper, they will slowly degenerate during a running simulation. That's because we modify an existing matrix in every Brush Cycle, and due to

float inaccuracy, errors will creep in. This will start stretching your starships even after just a few seconds of movement, as the orientation matrix (or better said the rendering of an object using a transformation matrix based on the orientation matrix) is very intolerant to even the smallest fluctuation. To make sure the matrix stays valid during the whole simulation, we will have to re-orthogonalize it in every Brush Cycle. There are numerous ways to do that. For the current Telesto build, I chose a very geometrical solution.

Re-orthogonalization of an orientation matrix	
Input	An orientation matrix, represented by its unit vectors: `float3` **`X, Y, Z`**
Output	An orientation matrix, represented by its unit vectors: `float3` **`X', Y', Z'`**
Algorithm	**`X'`** `= normalize(`**`X`**`)` **`Z'`** `= normalize(`**`X`** `×` **`Y`**`)` **`Y'`** `= normalize(`**`Z'`** `×` **`X`**`)`

Another note about the implementation is that Telesto does not store the Physics Object state (linear velocity, position, angular velocity, orientation) in the Physics Object directly, but in a separate state structure. There are, in fact, two state structures stored per Physics Object, representing the current and the last Brush Cycle state. This is necessary as collision analysis, as we will soon see, requires jumping back in time and subdividing the time in smaller steps as the Brush Cycle would do.

10.2.2. Collision Geometry

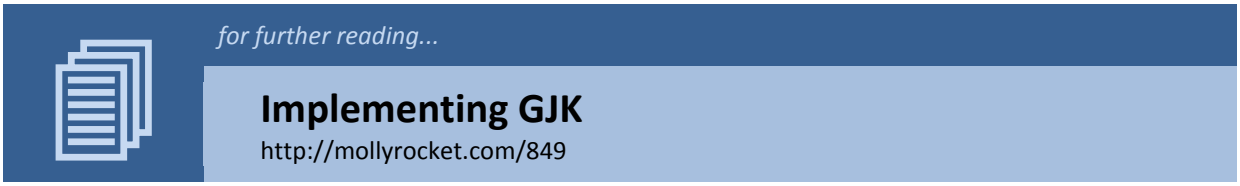
With the dynamics done, we may now accelerate a spaceship and push it around via forces, but it will still interpenetrate asteroid belts at will. In our real universe, the solidity of an object is an inherent property caused by sub-molecular forces. The in-game universe neither knows about atoms nor binding energy, so it is necessary to define different rules to mimic the solidity of real world bodies. This is the second important feature of a rigid body physics simulation.

It is best to begin with the very basics: the collision geometry. The requirements of the physics engine towards their geometry are completely different as the renderer requirements. The physics engine, for example, has to deal with many triangles, which could mean to - in a suboptimal approach - iterate over all triangles on the CPU, while the triangle count of geometry is almost irrelevant for modern graphics cards. The shader complexity is instead much more significant. It is therefore reasonable to not use the render geometry for physics simulations. Instead, a simplified geometry is created, following other constraints. A collision geometry must be fast to answer collision queries, which are queries whether or not two bodies penetrate each other, and if so, where. This is important to note: a collision algorithm usually does not check for collision, but for interpenetration. This is done to avoid separate handling for the many special collision cases, like edge-edge or even point-point collisions. It is also done to achieve a certain flexibility in processing, as a real collision situation would require finding the precise point of time between two Brush Cycles where both bodies touched, but did not penetrate each other. The collision response algorithm

usually expects a point of penetration and a normal for the collision surface, pointing towards the first query object by convention.

Two penetrating objects do of course produce an interpenetration volume, and not a point. Yet, all physics engines reduce a penetration situation to a single point and a normal for performance reasons. That adds a certain bias to the result, but physics geometry is just an approximation of the actual body either way, and the penetration situation is just an approximation for the collision situation. Adding another approximation, like replacing the penetration volume with a point, won't hurt too much. At the end of the day, a physics simulation for a game engine is a very simplified version of real physics, tuned for performance. This can be observed in many physics engines, like Havoc (41) as used in Half-Life 2 (6). If an object such as a crate falls on the floor, the collision might produce a response, as if one of the corner points hit the floor first. This happens, because many collision algorithms - especially for convex polyhedrons - reduce the collision volume to a corner point. But even in the real universe, there is an atmosphere between the crate bottom and the floor preventing perfect face-face collisions - not to mention that there aren't even precise faces on an object built of atoms.

The probably most common type of geometry used for physics queries are convex polyhedrons. They are often the technique of choice because there are efficient algorithms to achieve a collision point. The most popular algorithm is the Gilbert-Johnson-Keerthi distance algorithm (68), better known under its abbreviation GJK. I will not describe the complete algorithm in this document, but just shortly summarize the basic idea behind it. It builds up on the Minkowski addition of two bodies, A and B, which produces a Minkowski body M. The Minkowsi body M can be thought of as the body which is described by all points resulting from a pairwise subtraction of all points of the bodies A and B. If A and B were penetrating, there were some shared points which were both contained in A and B. Therefore, the Minkowski body contains the origin, as some of the point pairs contained two identical points, resulting in the null vector. The GJK algorithm now tries to find a polyhedron within the Minkowski body, which contains the origin. If one can be found, the bodies A and B are intersecting and vice versa. Telesto does not use the GJK algorithm for convex polyhedrons, but it is still a quite efficient implementation. There is a great video at Molly Rocket (69) explaining a brief implementation-oriented version of the GJK.



Telesto utilizes triangle geometry for the physics geometry. A triangle mesh is of course not necessarily a convex polyhedron. In fact, Telesto supports "slightly concave" bodies. This means that the algorithm itself supports concave geometry, but the result gets worse the more extreme a concave body is. A "C" shape is a good example for a very concave body, while a dent in a surface is a good example for a "slightly concave" element. Convex polyhedron algorithms have to divide a concave body into a group of convex polyhedrons. This can usually be done during load time. For Telesto, however, I wanted to save myself the option to do damage visualization by deforming geometry. This could lead to dynamic subdivision of concave bodies in convex polyhedrons, which is

a very complex and time intensive process for generic geometry. To avoid potential lags during the physics simulation, a triangle mesh geometry might offer better performance. Yet, for the non-deformable rigid bodies as they are used at the moment in Telesto, the convex Polyhedrons would probably be faster. The algorithms used for triangle meshes in Telesto are described in the following chapters.

10.2.3. Collision Detection

The first collision query to look at is the collision detection. It answers the question as to whether or not two objects are intersecting. The collision detection is called many times for a single pair of colliding bodies, as we will see in the collision response chapter - it is used to subdivide time to find a point of time as close to the moment of collision as possible. Due to this high frequency of calls, the collision detection requires a fast implementation. For now, let's start on the very basics: intersecting two triangles. There are countless algorithms to solve this problem as well as optimized implementations for them (70) (71) (72). Telesto uses a basic approach, as follows. I will not go into detail for this implementation and the math behind it, as it is a very well known problem.

Triangle-triangle intersection	
Input	Two triangles
Output	An interval (the start and end point of the intersection line segment)
Algorithm	1. Compute the plane quotations defined by the vertices of both triangles or their normal vectors 2. Compute the intersection line of both planes 3. Compute the line segment of the triangles on the intersection line 4. Project the line segments on the largest axis, compare the intervals and return output. Return no collision if the intervals do not intersect

A collision detection algorithm may terminate and return true as soon as one positive triangle-triangle test is found. However, a physics geometry can easily have several hundreds or thousands of triangles, which would lead to unacceptably slow computation times if every triangle-triangle pair has to be compared. Even worse, if no collision is found - the case that happens most - all triangle pairs have to be checked. This problem can only be solved by reducing the number of triangle checks required. This can be achieved by using hierarchical bounding volumes.

The idea behind hierarchical bounding volumes (also known as bounding volume hierarchies) is to generate a simple bounding geometry which completely contains the model. A first check is then performed on the bounding geometry. If the bounding geometries should intersect, their children are checked for collision. The children could either be triangles, or other bounding volumes. Consider your keyboard, for example. A very rough bounding volume could be a sphere that completely contains the keyboard. Spheres are popular bounding volumes - besides axis aligned bounding boxes and oriented bounding boxes - because their intersection checks are very simple. Two spheres intersect if the distance between their centers is less than their summed radii. Telesto uses spheres for hierarchical bounding volumes, too. As you move your finger towards your keyboard, you will intersect the imaginary bounding sphere around it. The bounding sphere is now split into its next pair of children - hierarchical bounding volume trees are usually binary trees. This

could be a sphere around your main key part and a smaller sphere around the numeric keypad. As you move your finger closer to the "T" key, the main-keyboard-sphere will be split in its children and so forth until you arrived at the very "T" key. Now the children are the two triangles forming the top of the key and an intersection is found. The following figure shows the process in 2D, with circles and line segments, as the hierarchical bounding spheres are hard to visualize in 3D. The algorithm itself works in exactly the same manner, but there are more efficient solutions for 2D situations. This example just serves the purpose of visualization. It demonstrates a line-segment-body and ray intersection. A body-body intersection would work just the same, but the hierarchical bounding volumes of both bodies would have to be split. I chose this illustration as it illustrates the tree traversal quite well.

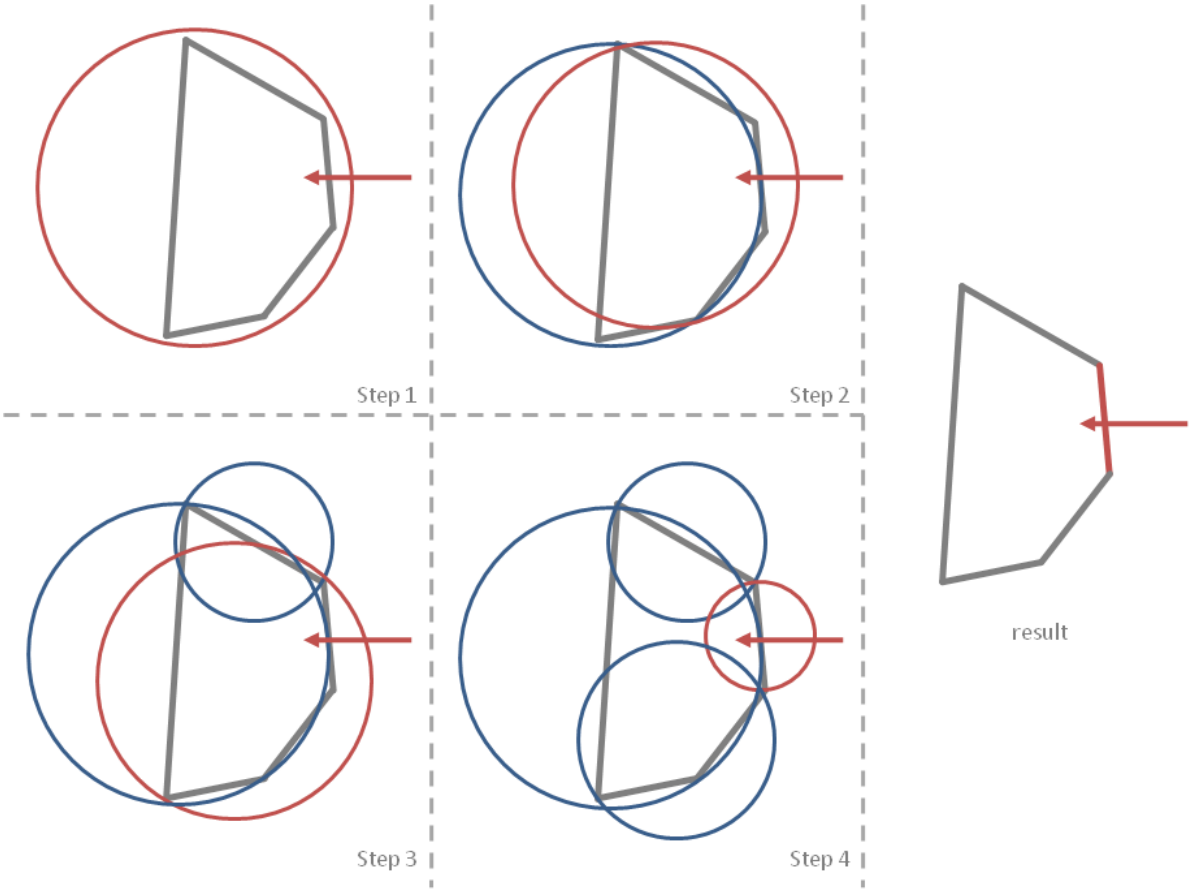


Figure 17: Bounding Volume Hierarchy sample

There are different algorithms to traverse the hierarchical bounding volume tree. In the example above, there are several spheres penetrated by the ray, but only the one closest to the ray origin is split. Other algorithms might just split the first sphere found. In addition, a bounding sphere may contain bounding spheres which actually reach outside their parent, as long as the actual triangles (or line segments in this example) are completely included in the parent. That's a feature of building the hierarchical bounding volume tree. Many algorithms will not follow this precision, but just enforce that a parent completely includes all its children bounding objects. I will take a look at the way hierarchical bounding volume trees are constructed soon. For now, I want to take a look at the algorithm involved in climbing through the tree in Telesto.

Telesto uses a hierarchical bounding volume tree composed of spheres with bounding volumes organized in a binary tree structure. The items of the tree are stored in a separate structure from the actual triangles, to minimize the data required to be shuffle around. A tree item consists of the bounding object, its child IDs and a triangle ID if it is a triangle. This means that even single triangles are represented by an enclosing sphere. This is done during the tree construction algorithm and has no major significance for the actual traversal algorithm. The tree is constructed during geometry loading and stored in object space, which means that the spheres have to be transformed for every intersection query. However, for a sphere, the only thing to do is to transform the center vector.

Telesto traverses the tree iteratively and always splits the tree item first, which is first checked. As the algorithm uses a stack for the loop, this behavior will cause the algorithm to traverse a chosen tree path first until there is either no intersection found on the path, or the triangles - the leaves of the tree - were reached. That's an important property of a hierarchical tree traversal algorithm, as otherwise it could start to split several bounding volumes of higher tree levels at the same time, while it would already have reported an intersection in the same amount of iterations used. The following algorithm is almost identical to the one used in the current Telesto build.

Hierarchical Bounding Volume tree traversal for a ray-mesh intersection check	
Input	The root item of the hierarchy `treeitem` **`root`** The transformation of the respective Physics Object `matrix4x4` **`t`** A ray for the intersection check `ray` **`r`**
Output	Whether or not an intersection could be found

Algorithm	`stack s`
	`s.push(`**`root`**`)`
	`treeitem i`
	`while not s.IsEmpty {`
	`  i = s.pop()`
	`  if i.IsTriangle then {`
	`  }`
	`  if TriangleRayIntersectionCheck(i,` **`t`**`,` **`r`**`) then {`
	`  return` **`true`**
	`  }`
	`  else {`
	`  if BoundingIntersectionCheck(i,` **`t`**`,` **`r`**`) then {`
	`  s.push(i.left)`
	`  s.push(i.right)`
	`  }`
	`  }`
	`}`
	`return` **`false`**

A small piece of code, but highly significant for the performance of the whole physics simulation. With this algorithm in mind, the benefits of a hierarchical bounding volume tree are obvious. If two bodies do not intersect because there is some distance between them, the first bounding volume intersection check will fail and the processing is done. If they intersect, the bounding spheres cause

it to identify the locality where the actual intersection takes place. This allows exclusion of a large part of the model within the first few iterations without even touching the triangles. Last but not least, sphere checks are a lot cheaper than triangle checks. For comparison, here are some benchmark results using different collision detection methods in Telesto in a few test scenes. The benchmarks were performed on the test system, A, as described in the appendix, with the same game mechanics and renderer modules running in all tests. Collision analysis and response used the same algorithm as the collision detection. The speeds show the average cycle rates after 10 minutes of simulation running in debug mode.

	Few collision scene	**Medium collision scene**	**High collision scene**
Triangle-Triangle Check	ca. 17 cycles/s	ca. 2 cycles/s	<1 cycle/s
Bounding Sphere Check	ca. 920k cycles/s	ca. 340 cycles/s	ca. 13 cycles/s
BVH Tree	ca. 1.3 Million cycles/s	ca. 1.1 Million cycles/s	800k cycles/s

In a low density collision test scene, a plain bounding sphere check before starting to do triangle-triangle checks helps to prevent doing checks for objects which are far away from each other. As the density increases, the bounding sphere checks will rapidly slow down, as more and more objects have to be compared. The only method with a good performance is a bounding volume hierarchy tree.

The efficiency of the hierarchical bounding volume tree, however, is highly dependent on the way the tree was built. A tree should be rather broad, in order to exclude as many parts of the model for collision checks as possible. A chain-like tree with leaves split over all layers of the tree could end with very unpredictable performance, as a slight change in positions of intersecting objects could lead to a much longer tree traversal time. As a general rule, a good bounding volume tree should contain its leaves in the last two or three layers. A suboptimal tree led to performance losses of over 70% in benchmarks.

How to build up the tree in the first place? There are three different methods. A bottom-up algorithm starts with the triangles and groups them together until only one element, the root, is remaining. This is probably the fastest algorithm, but might lead to slightly larger bounding spheres. A top-down algorithm, in contrast, starts with an initial bounding sphere over all triangles and then tries to split it into two parts. A top-down algorithm produces very tight bounding spheres, but might lead to longer processing times, as sphere splitting can be a difficult problem to compute. A top-down algorithm can be made a lot faster if not binary trees, but n-trees are used. Finally, trees can be constructed via insertion, by starting to construct bounding volumes on a certain section of the model and then continuing over the whole mesh, inserting new nodes in the tree. This allows one to best map the tree on complex geometry.

The current Telesto implementation uses a straightforward bottom-up implementation for tree construction. As a first step, bounding spheres of all triangles are constructed and put into a work set. A random bounding sphere B_1 is now chosen. All remaining bounding spheres are now searched for the bounding sphere B_2 which is closest to B_1. Once they are found, they are both removed from

the work set, and a parent is constructed containing both spheres and parent is put into a buffer set. The process continues with a new random bounding sphere, until the work set is empty (or there is only one element remaining which is then just moved into the buffer set). Once empty, the work set and the buffer set are swapped and the algorithm starts again, until only one sphere remains, the root item. This algorithm already does quite well, but optimization was added to handle bounding spheres containing other spheres.

10.2.4. Collision Analysis

Now we have a fast collision detection algorithm at hand. It is good to know which Physics Objects intersect, but in order to start a collision response procedure, we will also have to do a collision analysis to find a collision point and normal. To do so, we will utilize the same algorithms as we did for the collision detection. We will utilize the same bounding volume hierarchy and the triangle-triangle algorithm. However, this time, the algorithm will not terminate after finding the first intersecting triangle, but put them into a buffer to find all intersecting triangles. This will take longer, as we will have to traverse a larger portion of the tree. Once we find all intersecting triangles, we have to boil them down to a single point and normal. The collision point can be achieved by calculating the average of all collision points - the points on the line segment of intersection on each triangle in the buffer. The normal, however, should not be averaged. Instead, the average normal - weighted by the length of the intersection line segment - is used for a ray-cast at the averaged intersection point. The normal of the closest intersecting triangle is then used as the collision normal. This assures us that there is always a real surface normal passed to the collision response algorithm. Consider two cubes, penetrating at one edge. Taking a look at one of the two cubes, the "ring" of line segments will consist of a straight line of two faces of the cube. An averaged normal in this - admittedly slightly constructed - situation would lead to a more sphere-like collision, rather than a cube-like one. Something every observer will notice.

As we already described all the algorithms required to do collision detection, we are already done with the collision analysis and may advance to the next difficult part: the collision response.

10.2.5. Collision Response

The physics engine is now able to move objects according to their forces and torques and identify an intersection at a Brush Cycle. Still, nothing stops the objects from continuing to intersect. That's the part where the collision comes into play.

The collision response is triggered once the collision detection finds a pair of intersecting objects. In most cases, the actual collision would have happened somewhere between the current and the last Brush Cycle, and the dynamics simulation already moved the two objects so that they intersect. For a sufficient result, the collision response will have to find the instance of collision - or a point of time close to it. To do so, an iterative algorithm is pushed, subdividing the time between the current and last cycle to find a point of time close to the collision time, but still intersecting. To do so, the Physics Object states of the last cycle are required. The movement of both collision candidates is then interpolated and the time elapsed between both cycles is divided, similar to the bisection method. New collision detection queries are utilized until a predefined accuracy is achieved. The following figure illustrates the process for a preset maximum iteration depth of 3. The current Telesto build uses a depth of 5.

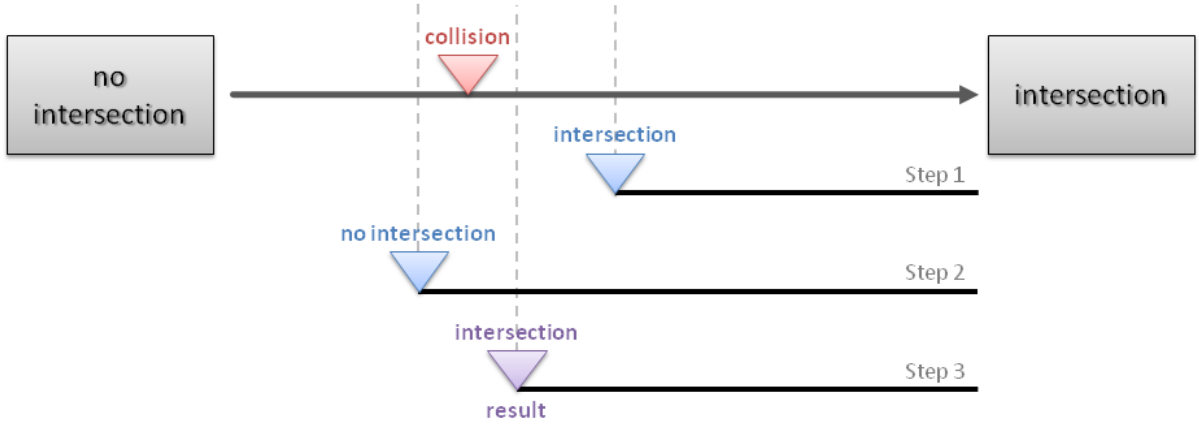


Figure 18: Time subdivision

Once the "almost collision" situation is received, the collision analysis is started with the transformations of the received "almost collision" situation. The resulting collision point and collision normal are then used for the actual collision response. The collision response now has to change the movement of the colliding objects, to prevent them from intersecting any further. Intuitively, one might suggest applying a new "collision force" at the colliding objects. However, a force is not a solution. A force would allow the objects to interpenetrate even further in the next cycles, as the influence of a force is applied over time. If you throw your keyboard at the floor, it will not slowly sink in the floor until it is accelerated back towards you. The collision response requires an instantaneous change of movement. To do so, a new magnitude is invented, the impulse. The impulse has no real physical counterpart. Yet that shouldn't shock us. The instantaneous change of movement in a collision situation is the result of molecular interactions and a lot of other details, which the physics engine cannot simulate. We will therefore make up the impulse to cover this important loss of detail. However, first things first. Telesto uses Newtonian Mechanics, so it's best to start with Newton's Law of Restitution. It relates the relative velocities between two bodies A and B before and after the collision. Magnitudes after collision are marked by a tilde in the following formulae. The normal is denoted as n in the following formulae and points, as noted in the collision detection and collision analysis algorithms, at A by convention.

$$\tilde{v}_{AB} \cdot n = -e v_{AB} \cdot n$$

This formula introduces some new variables. The coefficient of restitution (e) represents material properties of the bodies. A full elastic collision ($e = 1$) represents a rubber ball. For a space scene of metal starships, lower coefficients might produce a more believable result. The next construct introduced are the relative velocities between the bodies. At the instance of collision, both bodies share one point - or in the case of a game physics engine, we get one collision point from the collision analysis algorithm. The collision point is denoted as P. Even though both objects share the collision point, the velocities of the collision point for both objects can be quite different. Consider your finger hitting a keyboard key. The keyboard (object A) is stationary, so the key's velocity is zero. Your finger (object B), however, accelerates towards the key. At the moment of collision, the velocity of P on the key (v_{AP}) will still be zero, while the velocity of P at your finger (v_{BP}) is definitely higher. The relative velocity between your finger and the keyboard is given as follows.

$$v_{AB} = v_{AP} - v_{BP}$$

Substituting in the Law of Restitution results in the following formula, which will be our major workhorse for the following chapter.

$$(\tilde{v}_{AP} - \tilde{v}_{BP}) \cdot n = -e(v_{AP} - v_{BP}) \cdot n$$

We know the collision normal and the coefficient of restitution, which is often hardcoded in the physics engine or received from material properties of both colliding objects. Yet, the things we still miss are the velocities at the collision point. The velocity of a point on a body is of course dependant on the body's linear velocity. If a body moves with a certain speed without rotation, all its points have to move with the exact same speed. Rotation, however, might change the individual velocities. Consider a ventilator. While the object itself has a linear velocity of zero, its angular velocity still causes movement on the points of the object. This is expressed by the following formula for body A, body B works just the same. The vector o is the vector from the origin to the center of mass of the respective body - in other words its translation - and the vector p is the location of the collision point in world space. The formula may be used for both pre and post-collision velocities.

$$v_{AP} = v_A + \omega_A \times (p - o_a)$$

The pre-collision angular and linear velocity are known. Yet we have to find the post-collision versions of them. We already introduced the impulse. The impulse is a change in momentum and, as we ignore friction during the collision, it is completely directed in the direction of the collision normal (or in the opposite direction for object B). In this way, the linear velocities after collision can be written as follows, where j is a scalar defining the length of the impulse. Inertia tensors are considered in world space for the purpose of the collision response.

$$\tilde{v}_A = v_A + \frac{j}{m_A} n$$

$$\tilde{v}_B = v_B + \frac{-j}{m_B} n$$

$$\tilde{\omega}_A = \omega_A + j\, I_A^{-1}\, ((p - o_A) \times n)$$

$$\tilde{\omega}_B = \omega_B - j\, I_B^{-1}\, ((p - o_B) \times n)$$

These formulae may now be substituted into the formulae above, to form the point velocities of the collision point on both bodies before and after collision. These formulae are finally substituted into the Law of Restitution which may then be resolved for the length of the impulse j. This leads to an evening-filling formula to solve. Don't forget that the inertia tensor is a matrix, so pay attention to the order of multiplications. I will skip this part and present the result right away. The impulse length j may now be calculated and then be plugged into the four formulae above to build the collision response.

$$j = \frac{-(1+e)v_{AB} \cdot n}{n \cdot n\left(\frac{1}{M_A} + \frac{1}{M_B}\right) + \left((I_A^{-1}\left((p - o_A) \times n\right)) \times (p - o_A) + (I_B^{-1}\left((p - o_B) \times n\right)) \times (p - o_B)\right) \cdot n}$$

Most developers will just copy and paste the formula. Yet it is important to understand the details behind it to improve the physics model in the future, like adding friction to the collision with a tangential impulse or working with deformation of models. Telesto does a few calculations more than actually required and uses less optimization as possible to calculate the impulse, in order to keep the math and physics behind it accessible. The next builds will probably include a first framework for basic model deformation and perhaps friction for collision, although the latter is a difficult topic on its own. Both are not within the scope of this paper and would probably be worthy of their own thesis.

With the collision response done, we now have a simple but efficient physics engine, including dynamics, physics geometry and the collision handling algorithms. Yet, due to the fact that we made some assumptions and reduced the collision analysis to a collision point, we might still get interpenetration. That's true for most commercial-scope physics engines, too. To still generate a believable output, most physics engines use geometry which is actually larger than the render geometry, to add a "safety margin". Utilizing this feature, the Telesto engine already produces a fast and believable rigid body simulation for space-based scenarios.

The physics simulation presented so far works well on paper and for many in-game scenarios, but there is still an important situation where it might fail terribly: fast objects. Although the whole process is called "Collision Response", we never dealt with actual collisions, but mere intersections, received from intersection checks. These intersection checks during the Collision Detection and Collision Analysis are not continuously computed. Instead, the Physics Objects are moved every physics cycle and then checked for intersection. You could visualize that as objects teleporting for small distances instead of actually moving. If an object teleports far enough (moves fast enough) that it completely jumps through another object, the physics engine has no chance to recognize the actual collision, which would have taken place in the real universe. The process to still catch these collisions is called Continuous Collision Detection.

Continuous Collision Detection is often introduced by generating speed boxes. When doing so, Collision Detection does not check the actual mesh geometry, but a generated geometry representing the movement of an object - or at least the object bounding box, which is sufficient for Collision Detection, but will result in a larger amount of Collision Analysis jobs. The following figure illustrates the process with a speed box generated from bounding spheres in 2D for better visibility, but 3D works in just the same way (the 3D result would have been a capsule).

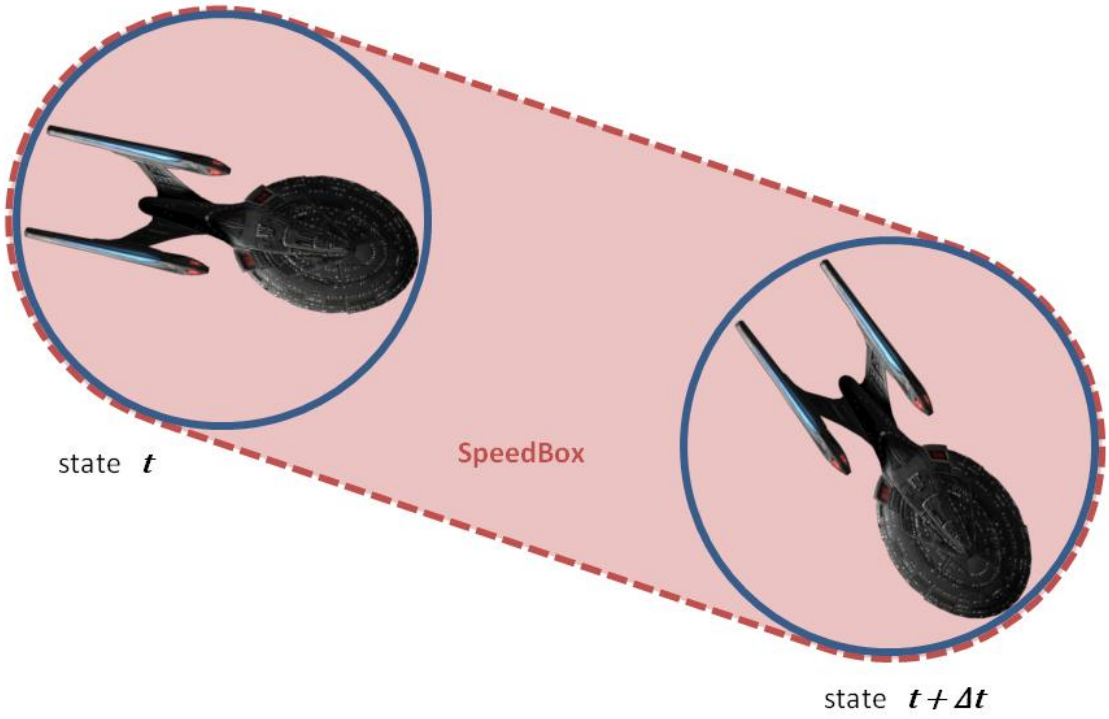


Figure 19: SpeedBox on Star Trek ships from Fleet Operations

Speed Boxes work well, and if a generic bounding volume is used, like a sphere, the generation and checks of a speed capsule are fast. A capsule is defined by two points and a radius, for example, which can be taken directly from the transformations of both states. However, the additional Collision Analysis tasks required when intersection speed boxes without an actual collision are generated will cost a lot more performance. Physics Objects will therefore attempt to use teleporting objects as much as possible. The vast majority of entities will do well with simple teleporting. Telesto performs a Brush Cycle at certain intervals, 10ms for example, which would lead to having an object move more than 100 times its length per second in order to actually miss a critical collision, such as passing through another object. That's quite unlikely for starships, if the speed of light is still the upper limit in the in-game universe. This is, however, very likely for bullets or laser pulses. Luckily, these objects are usually very small, and this additional knowledge may be exploited to do a very fast continuous collision detection algorithm. In other words, representing them as points and using raycasts. Telesto allows representing geometry as a point, which will lead to ray cast checks instead of the Collision Detection and Collision Analysis as described above. As points have no volume, points can't collide with other points. That's important for certain gameplay scenarios. Consider, for instance, an anti-missile system which destroys incoming projectiles by shooting them. More complex, larger objects will still require using speed boxes, like combat fighters, interceptable missiles, or comets.

10.3. GUI

The final sector of the Foundation I want to talk about is the user interface. As already mentioned in the input handling earlier in the Core, the user interface is split into two separate instances, the GUI itself, handling the placement of on-screen interface elements, and the GUI Logic, performing the actual processing behind the functions. The GUI Logic is nothing spectacular. It's just a Game Object with an LScript environment featuring event handlers for the different functions available to the GUI, like camera movement, activating an ability, or issuing a certain order. The GUI Logic is registered at the InputBrush, and Telesto offers support to register multiple GUI Logic objects to switch between different interface stages. This makes it easier to produce different controls, for example for the main menu to set up a game and for the in-game situation.

The GUI module is a bit more interesting. Similar to the Game Objects and Physics Objects, it contains GUI Objects. The GUI Objects handle their display on the screen and may link to a Game Object, which will then receive their InputEvents, like getting click, and may produce a GUIEvent, like firing a laser, in the input handler of the linked Game Object. A very simple button Game Object to handle the firing of a weapon on a GUI button click is given below. The `EventHandler` function expects a Game Object to bind the event handler to, the ID of the event-type the handler will listen too (remember that strings are actually mapped on IDs, so `"guiclick"` is actually a valid number) and finally the atom to body.

```
1  @guibutton
2  !EventHandler(~this, "guiclick", [
3     !GUIFire("fireblaster")
4  ])
```

The GUI objects build up all visible parts of the on-screen interface. They are arranged in a hierarchical system, starting at the GUIRoot, the only fixed and parentless GUI object, which covers the complete screen. All coordinates are expressed as offset between an anchor on the current object and a reference anchor on its parent. All GUI Objects are rectangles for the purpose of their management. Their graphical representation may of course be anything desired and the event handler, like the sample above, may filter received click events to mimic a non-rectangle button like a circle.

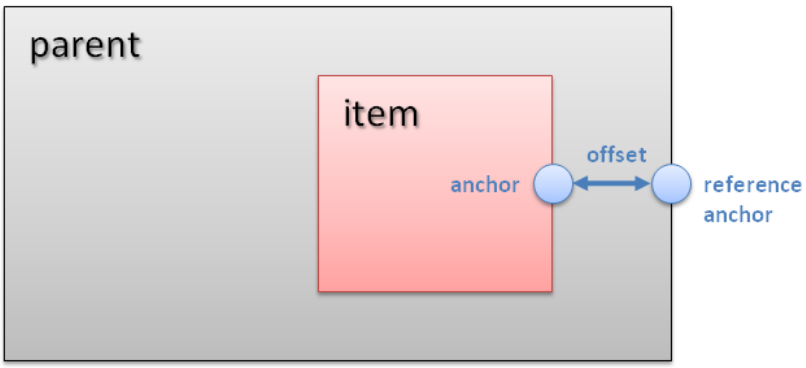


Figure 20: GUI Object Placement

The above figure shows an example of a "RightCenter" anchor and a "RightCenter" reference anchor. This is an efficient model to arrange GUI elements, as it works well with resizing an element. If the parent in the above figure were scaled to the right, the item will move along with it. To do so, a point-cast (strange name, I know) is performed on the hierarchy to return the relevant GUI objects for interaction. That's a rather fast process. The renderer will have to iterate over the GUI objects either way to refresh its transformations, so the offset-notation of GUI elements is no processing overhead for the externals, too.

Graphics-wise, the GUI objects support blending states. A GUI object may contain several states, which consist of offsets and a scale of the texture mapping. It is then possible to select two states and a blending factor via the GUI API, which allows soft transition between different GUI element states, like a soft on-hover and on-leave effect. As the blending is done in the graphics shader (well a renderer could of course treat it differently, but the recommendation is to blend the textures in the shader), all states have to be stored in the same texture, like in the following example, containing material for a Telesto Demo application I once made for Fleet Operations (18).

Figure 21: GUI Object Blend States

Similar blending could of course have been done by using multiple interface objects and animating their alpha color. However, as soft blending is a popular feature which is applied to almost all GUI elements in a modern game interface, I decided to do a direct implementation for it.

Another special element of the GUI is the Cursor. In the first implementations, I handled the cursor just via the GUI and GUI Logic modules, just like any other GUI object. However, that led to a slightly perceptible stuttering in cursor movement, as the distance between Safe Cycles - where the Input and therefore the mouse movement is being processed - is quite long in Telesto. To solve this issue, I created a separate cursor instance, which handles the coordinates of the cursor object. This cursor object is then directly fed from the LocalInput and the Renderer directly takes the coordinates from the cursor object for pure display reasons. The functionality of the cursor, like pressing a button or firing on-hover events are still only executed during the Safe Cycles, but the cursor now moves fluidly. Similar approaches are often titled "asynchronous cursor" in some game engines.

At the moment, the GUI has to be put together by hand; writing the script and definition files and placing objects via lambdas. The tooling for Telesto will of course also include an interface designer to offer simple drag-and-drop placement. At the end of the day, the handling of a GUI in functional languages is very intuitive, as you can just pass lambdas around to change the functionality of buttons. That's especially useful for elements that change often, like an action bar holding the abilities of a character in an MMOG.

11. Externals

External modules might include a renderer, debug analyzer, an agent gathering data for a web database or building the AI world. The most important concept for external modules is the extractor pattern, which I already described earlier. The remaining modules are not yet implemented in the current Telesto build or are placeholder implementations. The current renderer, for example, is just a straightforward DirectX11 rasterizer, supporting the complete Telesto API, but offering not much eye candy. I already have started working on a deferred DirectX11 renderer, which will appear in one of the next Telesto builds. A few new descriptions will probably be added too, for example in order to control post processing effects like bloom effects via LScript lambdas. The description and exchange-format workflow really ended up to be of great benefit for extending functionality later on.

I will not venture into the details of implementing a renderer, as that's clearly not the scope of this document and there is a ton of resource on this matter. With the externals left out for now, we are done with our first pass over the Telesto architecture.

12. Thread Map

A very important feature of a commercial-scale game engine is good performance. This is, naturally, closely linked with the hardware. With the current budget per CPU, the maximum calculation rate per second is stressing its physical limits. That's why multi-core architectures are becoming so popular in the past few years. A game engine will have to pay tribute to this development by supporting - and benefiting - from multi-thread architectures.

A very powerful solution to implement multithreading is a job system utilizing a Thread Pool. A Thread Pool is just a generic queue with several worker threads to process the incoming jobs. This offers good opportunities to adapt the Thread Pool to the current CPU, as the amount of worker threads may be increased or decreased without touching the actual implementations of specific features. The drawback is the completely asynchronous nature of a job system, which might be more difficult to manage and develop. The following figure shows a thread map of Telesto.

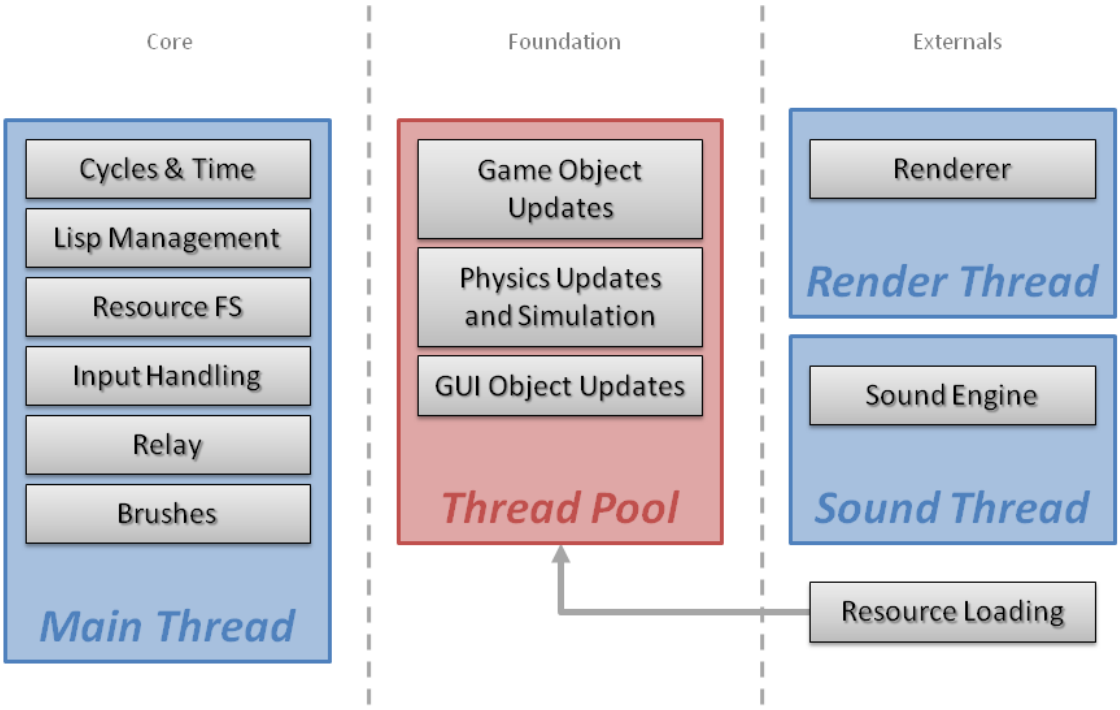


Figure 22: Thread Map

Telesto requires at least three threads, which produces sufficient performance on dual core systems. In this case, the Thread Pool is stagnating to a sequential processing of jobs in the main thread. Less than two hardware threads results in a significant performance loss, as a benchmark will soon show. Additional extractors, such as the AI, might utilize the Thread Pool. That's recommended for synchronous extractors, as the Thread Pool is rather unused in the synchronized window where no game mechanics or physics simulations are running. Asynchronous Extractors require their own thread, as that's the nature of an asynchronous module. Their resource loading could however be dispatched to the Thread Pool too.

Most Core functionality is located in the Main Thread. The basic universe runtime is established, the File System is generated, LScript compiler and string-to-ID mapping is performed and the important synchronous modules like the Relay and the brushes are running in this thread, too. Some modules

are not presented in the thread map, like the specific object containers, which deploy the jobs for the game mechanics and physics simulations onto the Thread Pool.

Working with multiple threads, especially a Thread Pool, requires having shared resources. This would be the case with the Game Objects for example. If shared resources are required, the engine will have to take care of locking. Conventional locking might result in unpredictable performance problems, due to threads falling asleep or - in the worst case - another process rising. A good pattern to work with locking is to minimize the time required to gather information from a shared resource. An implementation of this pattern is the indexing of variables in LScript like what was described earlier, where the lambda is able to ask for all required variables at once. If the access time on a shared resource is relatively short, a spinlock offers much better performance for short-term locks. It behaves like an "are we there yet? are we there yet?" conversation in contrast to the "not yet there? will ask again in 5 minutes" conventional lock. Telesto in fact uses just a single conventional lock, which is used in the initial setup phase. The following diagram shows some performance benchmarks made on different systems offering a different hardware thread count. The system details are available in the appendix.

	System A	System B	System C	System D
Idle Scenario	ca. 1.8 Million Cycles/s	ca. 1.0 Million Cycles/s	ca. 960k Cycles/s	ca. 220 Cycles/s
High Game Mechanics Scenario	ca. 1.3 Million Cycles/s	ca. 580k Cycles/s	ca. 620k Cycles/s	ca. 100 Cycles/s
High Physics Scenario	ca. 1.4 Million Cycles/s	ca. 620k Cycles/s	ca. 700k Cycles/s	ca. 90 Cycles/s
High Game Mechanics and Physics Scenario	ca. 1.1 Million Cycles/s	ca. 610k Cycles/s	ca. 580k Cycles/s	ca. 30 Cycles/s

13. Acknowledgments

I want to thank everybody who supported me in creating this thesis, especially my parents, Heide-Rose and Georg Stiegler, for their ongoing support and patience.

A special thank-you goes to Avery Russell for proof-reading my work and his support with the English language.

I would like to further thank my professors Walter Kriha and Jens-Uwe Hahn for their support and the great creative freedom they offered me for the creation of Telesto and this thesis.

Last but not least, I want to thank Bradford A. Smith, Harold Reitsema, Stephen M. Larson und John W. Fountain who discovered the Saturn moon Telesto on April 8th, 1980.

14. Integrity Statement

Herein I declare that this Master Thesis was created entirely by myself. I only used the sources and tools specifically stated in this document. Thoughts used, either by meaning or quoted, were marked as such.

Hiermit erkläre ich, dass ich die vorliegende Master Thesis selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Stuttgart, November 16th, 2010

_____

Andreas Stiegler

15. Appendix: Benchmark Systems

System A
Intel Core2 Quad (Q9550) @ 3 GHz
8 GB Ram
ATI Radeon HD 5870, 1 GB Video Ram, Driver version 10.9
Windows 7 x64

System B
Intel Core2 Duo (P8400) @ 2.26 GHz
4 GB Ram
Nvidia GeForce 9600M GT, 512 MB Video Ram, Driver version 258.96
Windows 7 x64

System C
Intel Core2 Duo (T6400) @ 2GHz
4 GB Ram
Nvidia GeForce 9600 GT, 1 GB Video Ram, Driver version 179.24
Windows 7 x32

System D
Intel Pentium 4 @ 3 GHz
1 GB Ram
Nvidia GeForce FX 5600, 256 MB Video Ram, Driver version 96.85
Windows Vista x32

16. List of Figures

Figure 1: Game Overview 21
Figure 2: Game Engine Overview 23
Figure 3: Constant Cycle Length 30
Figure 4: Variable Cycle Length 31
Figure 5: Causal conflicts during a cycle 33
Figure 6: Active Object with a private simulation 40
Figure 7: Active Object dependency problem 41
Figure 8: Modular Shared Memory Model 42
Figure 9: Behaviors 44
Figure 10: Early StarCraft alpha version, Blizzard Entertainment 50
Figure 11: Telesto Core Overview (Functionality) 57
Figure 12: Telesto cycle layout 58
Figure 13: Texture loading 71
Figure 14: Input Handling 75
Figure 15: Telesto Foundation Overview (Functionality) 77
Figure 16: Prediction 84
Figure 17: Bounding Volume Hierarchy sample 93
Figure 18: Time subdivision 97
Figure 19: SpeedBox on Star Trek ships from Fleet Operations 100
Figure 20: GUI Object Placement 101
Figure 21: GUI Object Blend States 102
Figure 22: Thread Map 105

17. References

1. **Szalai, Georg.** Video game industry growth still strong: study. *Reuters.* [Online] http://www.reuters.com/article/idUSN2132172920070621.

2. **Blizzard Entertainment.** *World of Warcraft.* 2004.

3. **Pettey, Christy.** Gartner Says "Generation Virtual" Will Have a Profound Influence on Culture, Society and Business. *Gartner.* [Online] http://www.gartner.com/it/page.jsp?id=545108.

4. **MicroProse.** *Sid Meier's Civilization.* 1991.

5. **Blizzard Entertainment.** *StarCraft.* 1998.

6. **Valve Corporation.** *Half-Life 2.* 2004.

7. **Crytek.** *Crysis.* 2007.

8. **Valve Corporation.** *Counter-Strike: Source.* 2004.

9. **Tripwire Interactive.** *Killing Floor.* 2009.

10. **Obsidian Entertainment.** *Neverwinter Nights 2.* 2006.

11. **Bugbear Entertainment.** *FlatOut 2.* 2006.

12. **NCsoft.** *Aion.* 2008.

13. **ArenaNet and NCsoft.** *Guild Wars.* 2005.

14. **Studio II Software GmbH and Ascaron.** *Sacred 2: Fallen Angel.* 2008.

15. **Nvidia.** *PhysX.*

16. **Nintendo.** *New Super Mario Bros. Wii.* 2009.

17. **1C:Maddox Games.** *IL-2 Sturmovik.* 2001.

18. Star Trek Armada II: Fleet Operations. [Online] http://www.fleetops.net/.

19. **Activision.** *Star Trek: Armada II.* 2001.

20. **Blizzard Entertainment.** *StarCraft II: Wings of Liberty.* 2010.

21. **Gas Powered Games.** *Supreme Commander.* 2007.

22. **Relic Entertainment.** *Homeworld.* 1999.

23. —. *Homeworld 2.* 2003.

24. **Ironclad Games.** *Sins of a Solar Empire.* 2008.

25. **Related Designs and Blue Byte.** *Anno 1404.* 2009.

26. **Friedmann, Thomas, Häuser, Thomas and Kneisel, Thorsten.** *Die Siedler II – Die nächste Generation.* 2006.

27. **Hidden Path.** *Defense Grid: The Awakening.* 2008.

28. **Pandemic Studios.** *Battlezone II.* 1999.

29. **DICE Schweden.** *Battlefield 2.* 2005.

30. **Pandemic Studios.** *Battlezone.* 1998.

31. **Namco.** *Pac-Man.* 1980.

32. **Paschitnow, Alexei.** *Tetris.* 1984.

33. **Nishikado, Toshihiro.** *Space Invaders.* 1978.

34. **MicroProse.** *Sid Meier's Civilization II.* 1996.

35. **id Software.** *Doom.* 1993.

36. —. *Quake.* 1996.

37. **Valve Software.** *Half-Life.* 1998.

38. **Epic Games.** *Unreal.* 1998.

39. **Blizzard Entertainment.** *Warcraft III: Reign of Chaos.* 2002.

40. **id Software.** *id Tech 6 Engine.*

41. **Havok.** *Havok Physics.*

42. Open Dynamics Engine. [Online] http://ode.org/.

43. **3Dconnexion.** *Space Navigator.*

44. *3Dconnexion.* [Online] http://www.3dconnexion.com/.

45. **Nintendo.** *Wii.* 2006.

46. FMOD. [Online] http://www.fmod.org/.

47. *Lambda the Ultimate.* [Online] http://lambda-the-ultimate.org/.

48. **Paramount Pictures.** *Star Trek.*

49. **Blizzard Entertainment.** *Warcraft: Orcs & Humans.* 1994.

50. *The Programming Language LUA.* [Online] http://www.lua.org/.

51. *SlimDX.* [Online] http://slimdx.org/.

52. *DirectX Developer Center.* [Online] http://msdn.microsoft.com/en-us/directx/default.aspx.

53. *XNA Developer Center.* [Online] http://msdn.microsoft.com/de-de/xna/default%28en-us%29.aspx.

54. *Microsoft .NET Framework.* [Online] http://www.microsoft.com/net/.

55. *Xbox.com.* [Online] http://www.xbox.com/en-GB/.

56. **id Software.** *Doom 3.* 2007.

57. *Notepad++.* [Online] http://notepad-plus-plus.org/.

58. *Autodesk 3ds Max.* [Online] http://usa.autodesk.com/adsk/servlet/pc/index?id=13567410&siteID=123112.

59. *Improving .NET Application Performance and Scalability.* s.l. : Microsoft.

60. *smalltalk dot org.* [Online] http://www.smalltalk.org.

61. *Haskell.* [Online] http://haskell.org/.

62. *OpenGL - The Industry's Foundation for High Performance Graphics.* [Online] http://www.opengl.org/.

63. *World of Warcraft Forums - How many KB's an Hour does WoW use?* [Online] http://forums.worldofwarcraft.com/thread.html?topicId=17616152297&sid=1.

64. **Blizzard North.** *Diablo II.* 2000.

65. **Valve Software.** *Counter-Strike.* 2000.

66. *Server Support Forum - Traffic eines Counter Strike Servers.* [Online] http://serversupportforum.de/forum/faqs-anleitungen/8273-faq-traffic-eines-counter-strike-servers.html.

67. **Hecker, Chris.** Rigid Body Dynamics. *chrishecker.com.* [Online] http://chrishecker.com/Rigid_Body_Dynamics.

68. **Gilbert, E.G., Johnson, D.W. and Keerthi, S.S.** *A fast procedure for computing the distance between complex objects in three-dimensional space.* 1988.

69. *Molly Rocket - Implementing GJK.* [Online] http://mollyrocket.com/849.

70. **Möller, Tomas.** *A Fast Triangle-Triangle Intersection Test.*

71. *Intersection of Triangles in 3D Space.* [Online] http://www.applet-magic.com/trintersection.htm.

72. **Tropp, Oren, Tal, Ayellet and Shimshoni, Ilan.** *A fast triangle to triangle intersection test for collision detection.* 2005.

73. **Varszegi, Jeff.** How to Write High-Performance C# Code. *dotnet.sys-con.com.* [Online] http://dotnet.sys-con.com/node/46342.

74. **Gregory, Jason.** *Game Engine Architecture.*

75. **Martin, Robert C.** *Clean Code - A Handbook of Agile Software Craftsmanship.*