Open World Streaming: Automatic memory management in open world games without loading screens.

Alejandro Juan Pérez

June 7, 2015

Contents

1 Introduction

Open world games are among the most appreciated games by players. They provide users the chance to explore a big world and they are very immersive. Open world games grant players the freedom to take any decision and enjoy their high interaction.

As you might already know, in open world games the player usually controls an avatar over a big virtual world. So the player that controls the avatar takes its skin and, ideally, gets immersed in the virtual world.

Sure you know about some of the most famous open world games. Some of them are: *Grad Theft Auto* series (by *Rockstar*), *The Elder Of Scrolls* series (by *Bethesda*), *Minecraft* (by *Mojang*). These games, apart from being very famous, have all been in the top of the best-selling videogames. Let's see some examples. *Minecraft* has sold 19 million copies for the PC platform and it is the most sold PC game ever. Grand Theft Auto V has sold 19 millons copies for the *PS3*, making it the most sold *PS3* game ever. The *Eder Scrolls: Skyrim* has sold 20 millon copies for *PC*, *XBOX* and *PS3*[1].

Therefore, open world games are not only about fun, freedom and high interaction but also a promising business. And as we will see they are technologically challenging.

Despite open world games are well know, the techniques that are applied in their implementation are not. Game studios that are experienced in the implementation of open world games, like *Rockstar*, keep well their secrets. The development of open world game engines requires a great investment.

The purpose of this project is to develop a game engine that will support making open world videogames. Our engine should allow the creation of games with huge worlds and avoid loading screens.

2 Requirements

So we are willing to make a game engine that supports the development of open world games. We are going to focus on the requirements that are more specific for open world games. And requirements that are common for most game engines will be left as secondary requirements.

2.1 Main requirements

We consider these requirements are the main target to accomplish.

- **Visualization:** We said that requirements that are common for any game engine? will be treated as secondary requirements. But this is an

exception. It is very important to have some kind of visual feedback. We need visualization to test our system. Also, it is important in order to prove that our engine is working properly. So we need to implement a graphical interface even if it is not very fancy. The graphics will be 2-dimensional.

- **Memory management:** If there is one thing that characterizes open world games is that they usually have very big worlds. These worlds have a huge amounts of data (entities, textures, meshes, sound, etc). Some of this games require several tens of gigabytes of compressed data in secondary disk. Obviously we can not expect our users to have that much RAM capacity. The main challenge will be to keep memory consumption to a minimum while not compromising playability or quality. The tick we are going to use is to load into main memory only the things that are closer to the player. Another thing that we will do is to avoid duplicated resources in RAM. For example, there are is a forest that has many trees that share the same set textures, we must accomplish that each texture is in RAM at most once.

- **Accurate physics simulation:** Again, this is a feature that most game engines incorporate. But it is important as a main requirement because accuracy in open world games carry challenges that must be solved. We will talk about this topic later because it is complex and requires its own space.

- **Easy to use:** It is important that making games with our game engine is as easy as any of the engines we are used to. If our engine was too difficult to use nobody would use it.

- **Testable:** We want to be able to see if our engine is working as expected.

2.2 Secondary Requirements

These requirements would be great to implement for a commercial game engine but not the main target of this academic work.

- **Sound:** Every game engine has sound and it is not an issue in open world games.

- **Scripting:** We could embed some scripting language as it is done in most game engines.

- **Advanced graphics:** They have nothing to do with world streaming.
- **Persistency:** It would by great if changes made to the world were persistent.
- **Efficiency:** Saving computational resources in world streaming will make them available for other tasks like physics simulation or artificial intelligence.

3 Level-based videogames

Traditionally games have been structured in levels. In this kind of games the player must complete the current level to advance to the next one. Once one level is completed that level doesn't need to remain in main memory therefore resources can be deleted. The mechanics of this type of games allows to manage the resources of the game in an easy and efficient way.

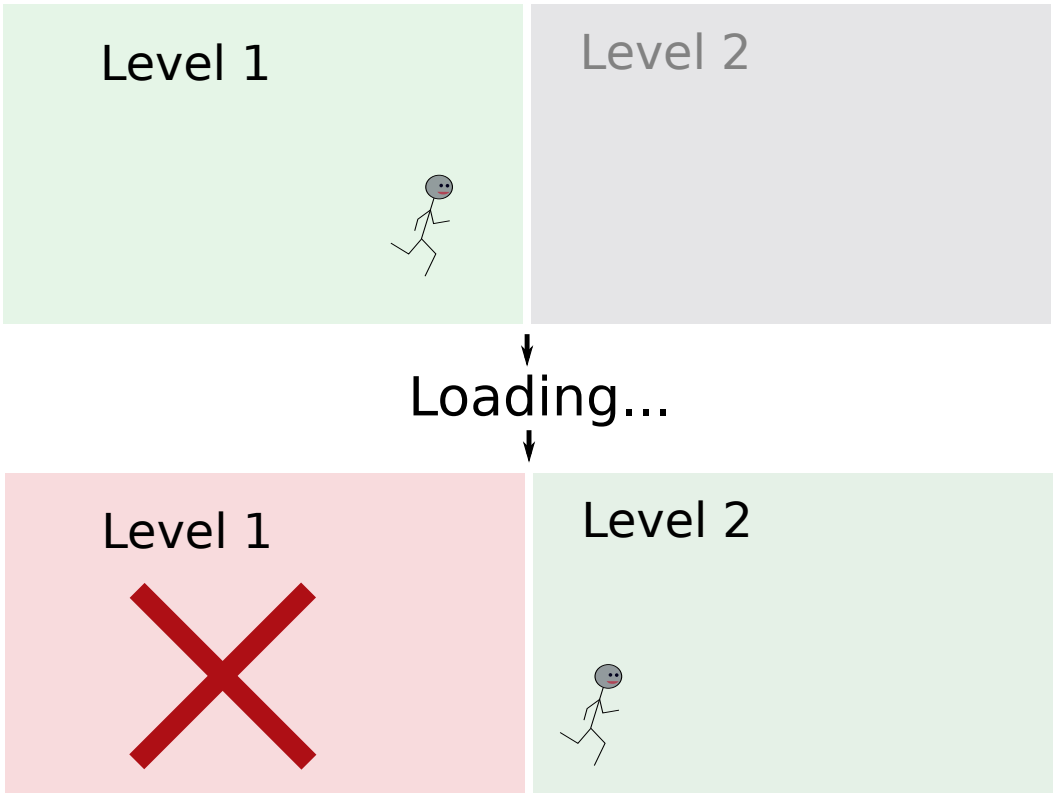

Figure 1: Level loading mechanics

It is well know that computers(and other gaming devices) usually have two types of memory. The first memory is the one we usually call main memory. Main memory is fast and slow. The second memory is the secondary storage memory. This one is slow and big. Changing of level requires loading all the resources of the next level from secondary memory and, therefore, it is slow. That's why in most level-based games changing of level will pop-up a loading screen.and the player has to wait for a while.

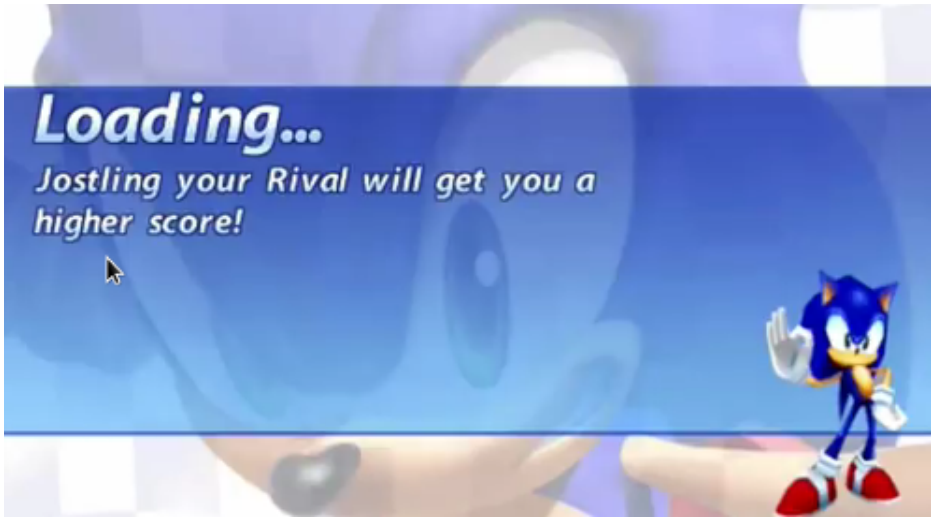

Figure 2: One loading screen from *Sonic*

Some times the designers of the game use the loading screen to give tips to the player.

The approach used by level-based games is not useful for open world games we need world streaming in order to avoid loading screens.

4 World streaming

World streaming consists on loading game resources on demand. The concept is very similar to video streaming. In video streaming the data is sent from disk or from the network as it is needed. So in world streaming the game contents are taken from the disk (or from the network) to main memory as they are needed. In contrast to video streaming, world streaming is not as straightforward. In video streaming the data is sequential, that is, you know what comes after. World streaming is different because what comes next depends on what the user does. So we must have the data prepared for all

the decisions the user can make. Imagine the case of platform game such as *Terraria*. *Terraria* is a 2D game that has an enormous map.

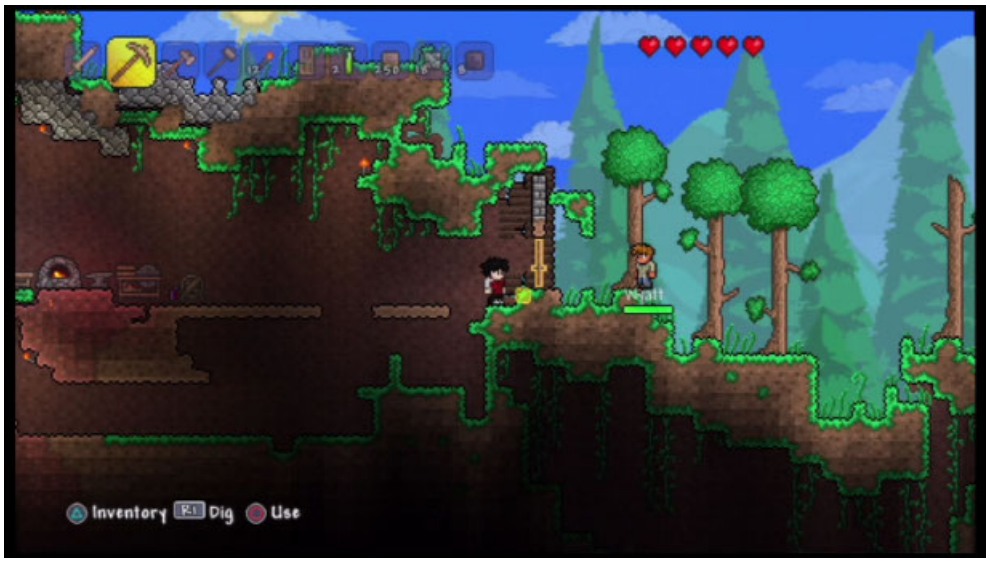

Figure 3: Screenshot of *Terraria*

In *Terraria* the player can move in four directions (up, down, left, right). So when the user chooses to go to any of the four directions the data must be already prepared in main memory no mater what key is pressed.

4.1 Examples of games using world streaming

"Streaming is the backbone of everything we do. Everyone at the company understands how it's structured." said Adam Fowler, the Technical director at Rockstar North [3]. Companies, such as Rockstar, know that world streaming is the technology that supports their success.

In this section we are going to show some of the most relevant games supported by the world streaming technology.

Hunter - Paul Holmes (1991) Hunter is one of the main influences of GTA (Grand Theft Auto). Hunter is a 3D action-adventure game in which the player could travel around a pretty big world. The degree of freedom in this game was enormous. There were many vehicles (bicycles, cars, ships, tanks, airplanes), places, and weapons. It is certainly not the first open world game but from the information we have I could affirm it might be the first game that used some kind of world streaming.

Figure 4: Screenshot of *Hunter*

In this game there were missions that could be completed but it was the player who decided whether or not to complete them.

Hunter was released for Amiga and Atari. The acceptance of the game was great and magazines ranked it with high scores.

Grand Theft Auto - Tarantula Studios (1997) This was the first Grand Theft Auto. With an enormous city and *mainly* 2D graphics, GTA was the one that started the famous series of violent games. The player takes the role of a criminal that can drive all around the city without loading screens.

Figure 5: Screenshot of *GTA*

The game was released in 1997 for PC and PlayStation, and in 1998 for GameBoy Color. The GBC (GameBoy Color) version is considered to be a great technological achievement due to the hardware limitations of the portable console. The first GTA was successful in sells mainly thanks to the GBC version but magazines and users rated it low[4].

Midnight Club: Street Racing - Angel Studios (2000) Midnight Club is a series of open world racing games. In this game you take the role of an urban street racer. The player can drive all around *The Big Apple* and challenge other street racers. In the missions you will have to defeat your enemies and scape from the cops. Completing missions will provide you respect and money to buy new cars. But you can skip missions and just enjoy driving around New York.

Figure 6: Screenshot of *Midnight Club: Street Racing*

This first Midnight Club was released in the same year that the PS2 (PlayStation 2) was commercialised. The acceptance of the public was good but sells were less than expected. One year later they released the GBA (GameBoy Advance) version which was rated poorly. Despite of its unfortunate sales this game is the one that started a successful saga.

Dungeon Siege - Gas Powered Games (2002) Dungeon Siege is a role-playing videogame very similar to *Diablo* series. Scott Bilas, one of the developers of the game, published a very inspiring article about how the team managed to develop a game with such a big world[2]. Thats to its flexible scripting engine the community was able to make modifications of the game, and even some people used the engine to make their own game. Despite the main story of the game is very linear there are many secondary missions and the player has quite freedom to move around. The greatest achievement of this game was avoiding all loading screens except for the initial one. Even taking portals is instantaneous. Graphics were impressive for the time being.

Figure 7: Screenshot of *Dungeon Siege*

Dungeon Siege had an online mode too. In the online mode you could grab your friends and complete the adventure in company.

Dungeon Siege had good sales but maybe not as much as it deserved for its technological quality and good graphics design

Grand Theft Auto: San Andreas - Rockstar North (2004) After the *Rockstar*'s successful titles *GTA III* and *GTA: Vice City*, it came an even more sold game: *GTA: San Andreas*. Take the role of *Carl Johnson*, the feared gangster. This game has one of the biggest maps in the history of videogames. There is complete freedom to wander around any of the tree enormous cities.

Figure 8: Screenshot of *GTA: San Andreas*

San Andreas is the most sold PS2 game ever. Very successful for the PC and XBOX platforms too. And not only that, the game is still having good sales for PC and Android.

World Of Warcraft - Blizzard (2004) *World of Warcraft* (*WoW*) is an MMORPG (massively multiplayer online role-playing game). This game has a different type of world streaming from the one we have seen so far. In this case the information that provides the status of the entities doesn't come from the hard disk but from the network. World streaming in the network is even more challenging but, for instance, resource management is very similar. This game has a big world and social interaction between players. Chatting, trading and fighting with other players is possible in *WoW*.

Figure 9: Screenshot of *World of Warcraft*

WoW has been the most played multiplayer game for many years and it is the game that provided most earnings of all times($10 billion)[5].

Fallout 3 - Bethesda Softworks (2008) World streaming is a feature present in many of the new-generation videogames. Fallout is just one of them that has an enormous world and impressive graphics.

Figure 10: Screenshot of *Fallout 3*

Minecraft - Mojang (2011) *Minecraft* is the sandbox MMO that proved that games are not all about graphics.

Figure 11: Screenshot of *Minecraft*

Minecraft is the most sold PC game of all times.

Conclusion We have seen many games supported by the world streaming technology. The trend shows that open world games are becoming more popular and it does not seem it is going to stop.

4.2 The strategy

The most valuable source of information I have found about world streaming is an article called *The Continuous World of Dungeon Siege* by *Scott Bilas*[2]. In this article, *Scott Bilas* explains how they developed *Dungeon Siege*. *Dungeon Siege* is an open world 3D game released in 2002. It was developed by *Gas Powered Games* and distributed by *Microsoft*. In this article I have found many useful tips for implementing an open world streaming engine. *Dungeon Siege* is a 3D game but you can only move in four directions (the world is landscape shaped). So the approach followed in *Dungeon Siege* is similar to the one we would follow in a 2D game.

In *Dungeon Siege* the world is divided into pieces of land which are aligned to a grid. These rectangles of land are called nodes. Any node can be connected to every other node. If two nodes are linked, that means that if the player is in one of them, he could travel to the other at any moment. Therefore, when the player is in one node we must be loading at least all the directly connected nodes. Normally the connections will match the adjacent pieces of land.

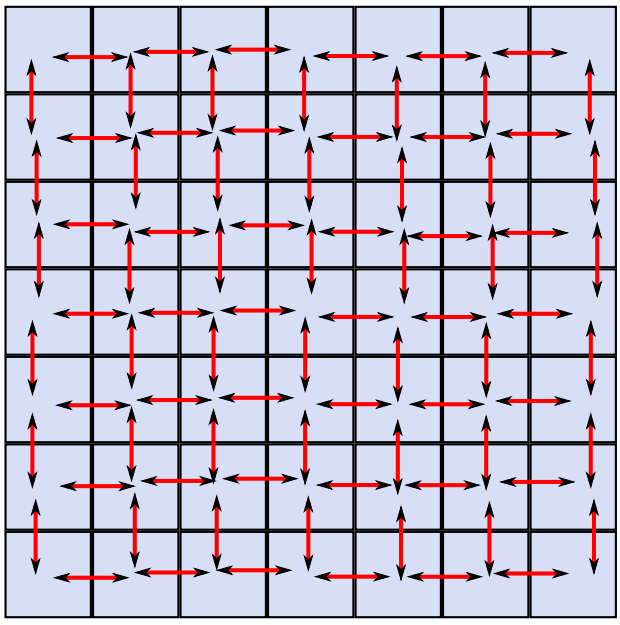

Figure 12: Connections of adjacent pieces of land (we are not taking into account the diagonals)

But there are cases in which two adjacent nodes could not be connected (e.g. there is a wall separating them). Or there could be nodes that are

connected and are not adjacent (e.g. there is a portal that will teleport the player from one place to another).

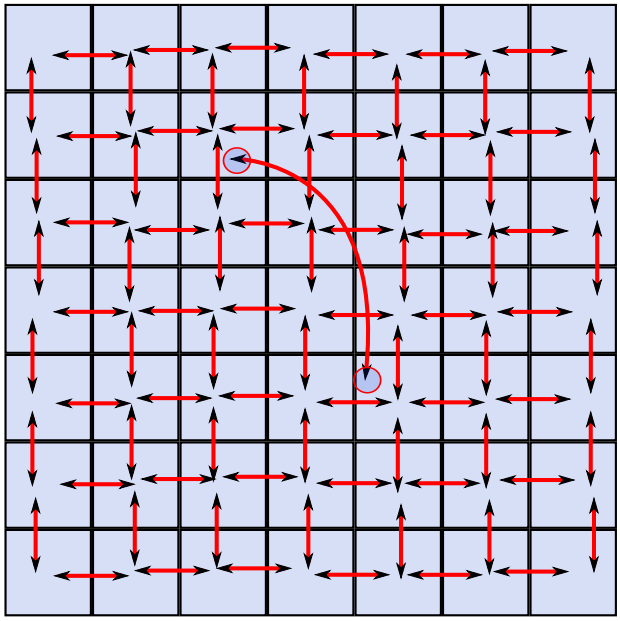

Figure 13: Two nodes connected by a portal.

This node-based approach is the one used in Dungeon siege. The designers of the game used a custom tool that would allow to manage the connections of the nodes.

For our engine we are going to take a simpler approach and we will assume each node is connected to all the surrounding nodes (i.e 8 nodes at most). This will simplify the job of the designers of the game. With our approach taking portals will require a loading screen (which is very common current games).

4.3 Loading Resources is slow

Imagine that we have implemented our strategy and in a given moment the player has changed of node. We will have to load to main memory all the entities that are located in the new adjacent cell (or node). That implies: loading from disk the file that describes that cell, parsing that file in order to find the entities, loading all the resources that are required by the entities and finally creating the entities. So when the node had been loaded you would realize that you have spent 1 second (or much more) and in this time

you haven't rendered a single frame!. Therefore, we have to solve this in some way.

The bottleneck here is the access to disk. Access to disk blocks the CPU and requires some time. So most of the time, when loading a new cell, the CPU would be idle while it could be doing important tasks (e.g. physics simulation or rendering). The solution that is suggested in the article of *Scott Bilas* (and the one we have followed) is to use a separate thread for loading resources from disk. So if our main thread (the one that executes the main loop) needs to load a resource, instead of doing it itself, it asks the background thread to do it. This way, the main thread is never idle when there is job to be done and the secondary thread will be loading from disk at its own pace.

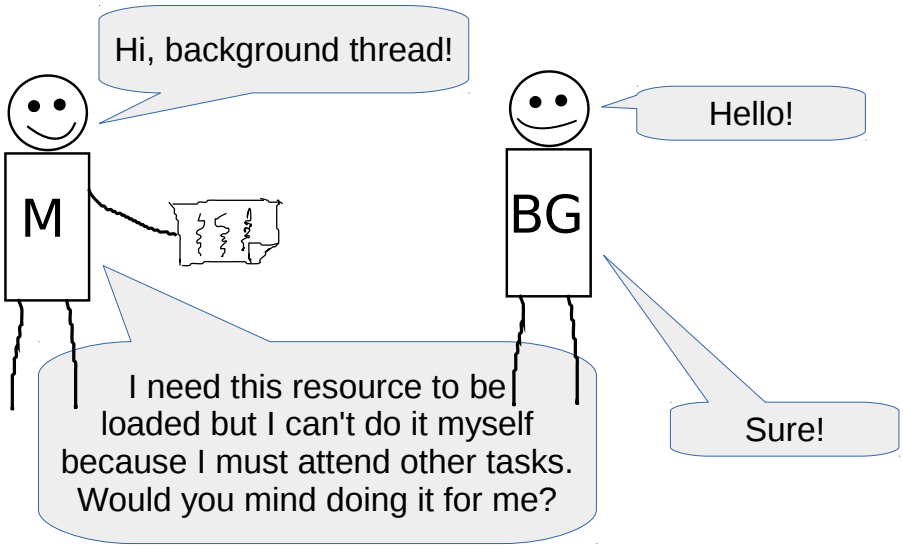

Figure 14: Two threads talking.

5 The architecture of the engine

Our game engine will be composed by some subsystems. In order to work properly the components of the engine must interact with each other in some way.

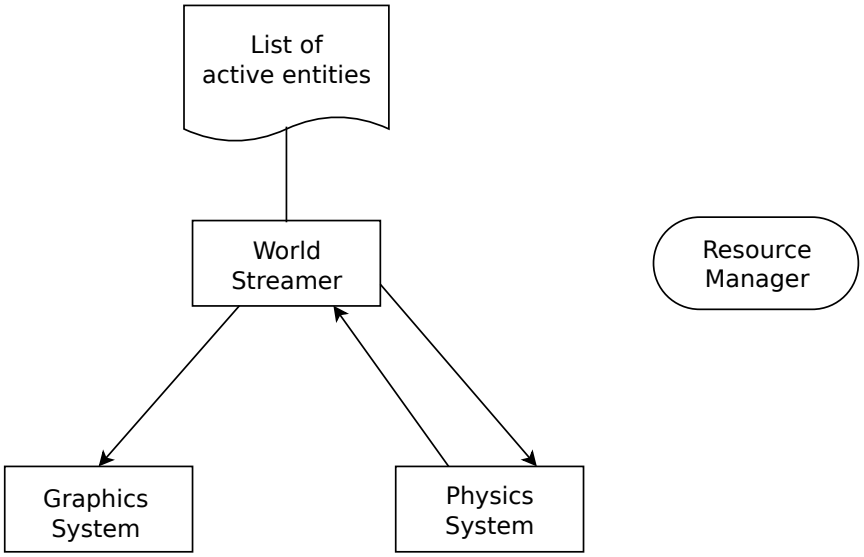

Figure 15: Game engine architecture.

The picture above is a simplification of the architecture. Notice that in the diagram only two subsystems are represented. The graphics and physics components are the most important subsystems, but there could be others like sound or artificial intelligence (AI).

Resource Manager: The resource manager is the component in charge of accessing the file system. It runs on a separate thread so it does not interfere in the progress of other tasks. Also, it must assure that there will not be duplicated resources in main memory. The resource manager is used by all the other architecture components. So it takes the role of a servant for the others. In other to ease the access to this utility system, it is very convenient to implement it as a singleton.

World Streamer: It is the one. The brain of the architecture. The world streamer is in charge of telling the subsystems below (graphics and physics in our diagram) what to do. It must decide whether a given entity must be loaded or not. An inefficient implementation of the world streamer would decide that all the entities of the world must be loaded. And a useless implementation of would not load any entity. So the world streamer needs to compute the subset of entities that must be loaded at any time. This subset of loaded entities is what we find in the diagram as "List of active entities".

Graphics System: Might be called graphics engine too. It will get a list of entities that must be rendered from the world streamer. So it will be just told what should be displayed on the screen. The graphics system in independent from all the other components and does not care who is using it.

Physics System Very similar to the graphics system. In the diagram you can appreciate the difference between these two: an extra arrow. This arrow flows from the physics system to to the world streamer. The meaning of this arrow is the position of the main character. After each physics simulation step the position of the main character might have changed and the world streamer needs this information in order to recompute the set of active entities.

6 Implementation

In this section I will explain briefly how I implemented the modules of the game engine.

6.1 Graphics System

The graphics system is implemented from scratch (there is a reason that will be explained later). Since implementing a graphics engine is not the purpose of this academic work, it does not have a big set of features.

6.1.1 Architecture

This graphics module is split in two layers. The first layer is the low-level layer. The low-level layer is the one that uses the OpenGL API (application programming interface). The purpose of the low-level layer is to provide a simple interface for the upper layer. This way the high-level layer does not need to understand any of OpenGL. So if we needed to deploy the application to a platform were DirectX performs better, we only would need to replace the low-level layer and the rest of the whole system would remain the same.

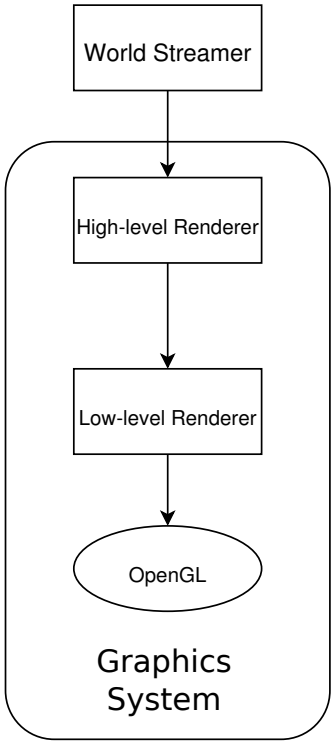

Figure 16: Graphics system architecture.

The low-level renderer API is just a reduced set of functions that allows to draw things on the screen. On the other hand the high-level renderer provides a class-based mechanism to define the scene. That is, in the low-level renderer we would say: "draw texture A in position X" and it will be shown on the screen for just this frame. And in the -high-level renderer we would say: "there exists a sprite in X and it has priority P" and it would be drawn on the screen until it is removed.

6.1.2 Features

The renderer I have implemented is not very sophisticated but works fine for our purpose. Animations are not yet supported. You can assign priorities to the sprites in order to define what is drawn on top. There is an abstraction of the camera too. The low-level renderer uses OpenGL ES (embedded systems), so it could be ported to mobile platforms.

6.1.3 Why did I implement my own graphics engine?

There are three reasons I had to implement a custom graphics engine.

Most graphics engines will manage resources automatically. Automatic resource management is one of the main targets of this academic work. If we had the render engine to do this for us, we would be missing something important. Even though, the graphics engines only manage graphics related resources and we would like to have resource management unified. What is more important, the resource manager of the graphics engines might be blocking.

OpenGL implementations do not support multithreading. OpenGL does not allow you to call its functions from threads other than the main thread. If you try to do so the behavior will be completely undefined [6].

Loading a graphics resource (like a texture) involves these steps:

- **Load the resource, which is stored in hard disk, to main memory.** This must be done by the secondary thread because we are accessing hard disk and that is slow.
- **Allocate the resource in video card memory.** To render an object all its resources must be in the memory of the graphics card. To allocate, for instance, a texture in video memory we need to call OpenGL functions. That is the reason this step must be performed by the main thread (only the main thread is allowed to call OpenGL functions). Fortunately, the process copying a resource from RAM to VRAM if fast enough, so it does not block other tasks.
- **Delete the resource in MM.** Once the resource is en VRAM we do not need it anymore en RAM. This task is not costly and could be performed by any thread. In our implementation it is deleted by the secondary thread because we think it makes more sense that the memory is released by the same thread it was allocated by. Also, in this way the code seems better encapsulated.

Most graphics engines provide functions to load graphics resources. And these functions do all the steps that we mentioned in a single call. Therefore, if we used one of these graphics engines, we would not be able to split the work among the two threads. Probably, there are workarounds that would allow to split tasks but they might difficult to implement. This is the main reason I have decided to make my own render engine.

I had personal interest in learning modern OpenGL. Also, I wanted an excuse to learn modern OpenGL features like shaders. The version of OpenGL I used is OpenGL ES 2.0. This version of OpenGL is compatible

with embedded systems like smartphones. In this version you are forced to use shaders because all the fixed pipeline functionality has been removed. I will dedicate a section to explain briefly how to use shaders in modern OpenGL.

6.1.4 Modern OpenGL programming

I do not pretend to make a tutorial on modern OpenGL nor shaders but just to summarize what I have done. I have used only the subset of API that is common to OpenGL 2.1 and OpenGL ES 2.0. In my testings, I used version 2.1, which is the one that runs on PC.

Since OpenGL does not support fixed pipeline functions I had to use shaders. What are shaders? Basically , they are pieces of code that run in the graphics card. They are used to achieve custom visual effects[7]. In OpenGL shaders are written in a specific programming language called GLSL. GLSL is very similar to C[8]].

There are two types of shader.

- **Vertex shader.** Vertex shaders take as input one vertex and gives as output the position that vertex should take. It is used to achieve effects such as mesh deformation. One example where vertex shaders are used is the waves of the ocean. The vertex shader should change the position of vertices so it fakes the water moving. In our 2D render engine, we do not need the vertex shader to do anything special: it just forwards the position as it is.

```
#version 120

/**
 * In the vertex shader you are computing the position
 * of the current vertex.
 * Also you compute the texture coordinate corresponding
 * to this vertex.
 **/

// these are the parameters received by the main program
attribute vec2 inPosition;
attribute vec2 inTexCoords;

void main()
{
```

```
        // compute the position of the current vertex
        // In this case you are just forwarding the position,
        // but this allows you to achieva cool effects like the
        // mesh bending or the waves of the ocean
        // The fouth coordinate is called W(1.0) and it is used
        // for normalization purposes( read:
        // "http://stackoverflow.com/questions/2422750/
        // in-opengl-vertex-shaders-what-is-w-and-why-do-i-divide-by-it" )
    gl_Position = vec4(inPosition, 0.0, 1.0);

    // compute the texture coordina corresponding to this
    // vertex
    gl_TexCoord[0] = vec4(inTexCoords, 0, 0);

}
```

- **Fragment shader.** Also called pixel shader but this term is more used for DierectX. Fragment shader is executed for every pixel and it should output the desired color for the current pixel. In our code we will be returning the color of the corresponding texture coordinate.

```
#version 120

/**
 * In the fragment shader you are computing
 * which should be the color of the current pixel
 * ( fragment and pixel are sinonyms in OpenGL,
 * in fact in DirectX it is called pixel shader )
 *
 * The output goes to -> gl_FragColor
 *
 **/

uniform sampler2D tex;

void main()
{

        // take the color of the texture pixel
        vec4 color = texture2D(tex, gl_TexCoord[0].st);
```

```
        // output
        gl_FragColor = color;

    }
```

References

[1] List of the best-selling videogames.
http://en.wikipedia.org/wiki/List_of_best-selling_video_games#All_platforms

[2] The Continuous World of Dungeon Siege - Scott Bilas.
http://scottbilas.com/files/2003/gdc_san_jose/continuous_world_paper.pdf

[3] Technical leads Adam Fowler and Phil Hooker take us through the technology powering GTA V.
http://www.develop-online.net/studio-profile/inside-rockstar-north-part-3-the-tech/0184140

[4] GTA article at Wikipedia.
http://en.wikipedia.org/wiki/Grand_Theft_Auto_%28video_game%29

[5] Top 10 highest grossing videogames of all time.
http://www.businessinsider.com/here-are-the-top-10-highest-grossing-video-ga
op=1

[6] Stack Overflow post discouraging multithreaded rendering with OpenGL.
http://stackoverflow.com/questions/11097170/multithreaded-rendering-on-opengl

[7] Shaders on Wikipedia.
http://en.wikipedia.org/wiki/Shader

[8] GLSL on Wikipedia.
http://en.wikipedia.org/wiki/OpenGL_Shading_Language