FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Geospatial Rendering in Unity 3D

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Zaufl Stefan
Matrikelnummer 0925357

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Wimmer Michael
Mitwirkung: Dipl.-Ing. Dr.techn. Christoph Traxler
Dipl.-Ing. Dr.techn. Gerd Hesina

Wien, 08.09.2014 (Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Geospatial Rendering in Unity 3D

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Zaufl Stefan
Registration Number 0925357

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Wimmer Michael
Assistance: Dipl.-Ing. Dr.techn. Christoph Traxler
Dipl.-Ing. Dr.techn. Gerd Hesina

Vienna, 08.09.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Zaufl Stefan
Wiethestraße 69/38, 1220 Wien, Austria

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Kurzfassung

Der Inhalt dieser Arbeit beschreibt die Entwicklung eines Terrain-Renderers unter Verwendeung der Unity 3D Spiele-Engine. Im Zuge dieser Arbeit werden wir einige theoretische Hintergründe erklären, aber uns hauptsächlich mit unserem Lösungsansatz beschäftigen. Wir werden die Implementierung anhand von UML-Diagrammen im Detail beschreiben. Am Ende werden wir Benchmarks diskutieren und analysieren welche Schwierigkeiten aber auch Vorteile durch die Verwendung von Unity 3D aufgetreten sind.

Abstract

The goal of this work was to create a terrain-renderer using the Unity 3D game-engine. We will explain some of the theoretical background, but will mainly focus on explaining our approach. We will discuss the resulting application in detail using UML-diagrams. At the end we will discuss some benchmarks we made and analyse what the advantages and disadvantages of using Unity 3D had been.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The company VRVis Forschungs GmbH [21] wanted to test if the Unity 3D [16] game engine can handle geospatial terrain-data (including their own data-format OPC). As Unity offers some extra functionalities like built-in shadow-support, collision-detection and cross-platform compatibility the company is interested in this engine. The aim of this work is to check whether Unity is a suitable engine to use, but not to create the best terrain-renderer possible.

1.2 Methodological Approach

The chosen approach for this work was prototyping. Throughout the development, working prototypes would be created, assessed and then a new one would be planned. Each prototype has its own focus, for example the first prototype would just aim at loading a single file and displaying it. At the end the prototypes would get more complex and include features implemented in their predecessors.

1.3 What is Unity

Unity is a 3D game-engine that works especially well with rapid prototyping. The killer feature of Unity is its "write once" mentality. It allows you to write code and compile it for different platforms like PC, Mac, mobile phones or gaming consoles. It also allows you to hit the play-button any time you want and change the parameters of your game-objects whilst in-game.

Unity is distributed under a proprietary license. There are 2 available licenses: Free and Pro. The free distribution lacks features for advanced effects (like render to texture) and shows a splash-screen every time you start your game-distribution. The Pro-license must be bought, but enables the advanced features and has no splash-screen [17]. There is also a license that gives access the engine's source code [18].

1.4 The Work's Structure

In this document, some theoretical backgrounds will be discussed in chapter 2 and related work is presented. In chapter 3 the methods we used to implement the terrain-renderer will be described. In the 4th chapter, the implementation itself will be discussed in detail. Chapters 5 and 6 will reflect on the work, summarize it and give a brief vision of what should be done next.

CHAPTER 2

State of the art / analysis of existing approaches

2.1 Literature Studies

Thatcher Ulrich discusses way of handling LOD (Level Of Detail) [15]: The maximum amount of detail that can be observed by a viewer ranges from a few millimetres at the tip of his toe to a massive mountain in the distance. To cope with this wide range of resolution he introduced a chunk-LOD system that picks an appropriate LOD-level for a single chunk based on the distance of the chunk to the viewer. Chunk-LOD-levels can be compared to mipmap-levels of textures: they have fewer polygons at higher levels, level 0 is the original level. When the viewer moves the LOD-levels change dynamical - to prevent popping the algorithm interpolates the vertex positions between the levels. The other problem is that cracks might occur at the boundary of chunks at different LOD-levels. To solve this, Thatcher adds new triangles to the boundary of a chunk that fall like a curtain and block any cracks.

Frank Losasso and Hugues Hoppe introduced geometry clipmaps [6] in their paper - a method for rendering different LOD-levels: The viewer is surrounded by a set of nested regular grids that each contain a different LOD-level. When the viewer moves, these grids get incrementally refilled, which can be done quickly, because they are stored as vertex-buffer in video-memory. The advantage of that method is that no popping occurs(visual continuity) and that the frame rate is uniform. They also describe a method that compresses terrain data(in their example they had a 40GB dataset compressed by the factor 100).

One year later, Arul Asirvatham and Hugues Hoppe developed a GPU-driven version of geometry clipmaps [1]. They introduced an algorithm that processes all informations on GPU enabling them to render a 20 billion sample-grid with 90 frames per second using 355 MB of memory.

Filip Strugar [10] states that it would be more wise to use the GPU to create the needed meshes rather than sending it from the CPU to the GPU, so he developed the “Continuous Distance-Dependent Level of Detail”-algorithm and made an reference-implementation of it.

His algorithm tries to keep the on-screen triangle complexity equal. He uses a quad-tree of heightmaps to select appropriate LOD-levels and uses the vertex shader to morph from one LOD-level into another. Because all this is one big mesh, no mesh stitching occurs or curtains are needed. This algorithm takes the 3D-position of the camera into account: this means that the LOD-level also changes when the camera moves up and down.

Thomas Ortner, Gerhard Paar, Gerd Hesina, Robert F. Tobler and Bernhard Nauschnegg faced the challenge of rendering the inside of tunnels with a resolution of 1mm. They obtained their data via laser-scans - because of the huge amount of data they produced they had to implement a potent Level of Detail system. They grouped data into so-called OPCs and only kept visible groups in memory. Each group can be represented as a quad-tree with different levels of detail. [11]

2.2 Analysis

Modern papers focus more on algorithms that do most of the work in the graphics card. This is very likely caused by the growth of computing power on the graphics card and its parallelism. As stated in Filip Strugar's paper [10] the computing power of graphics cards increased more than the power op the CPU. He also says that even a simple GPU-algorithm can produce better results than the most sophisticated algorithm on the CPU. The work of Arul Asirvatham and Hugues Hoppe [1] also seems to support that trend, because they developed a GPU-based approach of geometry clipmaps with better results than the original algorithm.

In contrast Thatcher Ulrich [15] presented a CPU-heavy algorithm in his paper from 2002 that does produce good results. But he has to introduce extra solutions to make smooth transitions from one chunk to another if they have different LOD-levels. It seems more logical to push the research more into the direction of GPU-algorithms, because of the parallelism of the GPU as suggested by Filip Strugar [10].

2.3 Comparison and Summary of Existing Approaches

Terraland

Terraland is a commercial tool that allows users to import geospatial data into Unity and display it as a mesh with the correct texture on it. You can download height- and texture-data directly from NASA within Unity with the TerraLand terrain module. Therefore, you have to select a specific region by entering the desired coordinates. With the TerraLand Earth module you get a Google Earth like representation where you can freely explore the planet. The Demo on their website also shows some flaws of the software: Although no cracks are visible between the patches, the user can clearly see popping artefacts occur when the LOD-level of a patch changes. [13]

HeightmapFromGridFloat

In the Unity-community there is a free alternative for directly loading height-data into Unity: The HeightmapFromGridFloat-script. This script reads a USGS GridFloat topo file which contains a stream of floats that represent the height of the terrain on a regular grid. The script completely lacks the support for texture and/or LOD. [8]

The Common Way

The most common way to import and display geospatial data into Unity seems to use 3rd-party tools like Terragen [7] or Blender [5] to create heightmaps for Unity and render them via the build-in terrain-object. [7]

2.4 Shaders in Unity

To give the viewer a better overview of what can be done when using Unity this section describes shaders in Unity. In Unity, a shader has to be assigned to a material and this material can be assigned to a surface. These shaders are written in Unity's shader-language: ShaderLab [9]. The real shader-code is embedded in ShaderLab and can be a surface shader(provided by Unity), a vertex and program shader written in CG [3] or a fixed function shader.

Some advanced techniques are not available for all platforms. For example, tessellation shaders in Unity are only available on the Windows platform, because it uses DirectX11. You can find examples for such a shader on the Unity website [14].

2.5 Level of Detail

When handling very large amounts of data, we need algorithms that help us cope with these amounts. Due to the limited amount of RAM and computational power we have to introduce levels of detail (LOD) to our application.

Robert Oates discusses an application in Unity that implements such a system [12]. The first thing he points out is that one cannot load the whole world into memory, but has to divide it into so called "chunks". A chunk is a piece of land with its own coordinate system. All chunks shall be of equal size. When a viewer looks into the world, the application creates an island of chunks around him (e.g. 9 chunks where the viewer is located in the middle one). When the viewer moves in one direction the chunks the player is moving away from get unloaded, the whole world is shifted into the direction of the unloaded chunks and at the other end of the world the same amount of chunks that were unloaded, gets loaded (e.g. from disk, see figure 2.1).

This method has the advantage of constant memory usage, because only a fixed amount of chunks is present in the memory at a time. The chunk shifting is necessary to prevent numerical errors from happening. In Unity transforms are represented using float-variables, which tend to get very imprecise at a value of approximately 20,000. When chunks are always located around (0|0), numerical errors are not visible. This technique is also known as terrain-streaming as discussed in the presentation by Robert Oates [12].

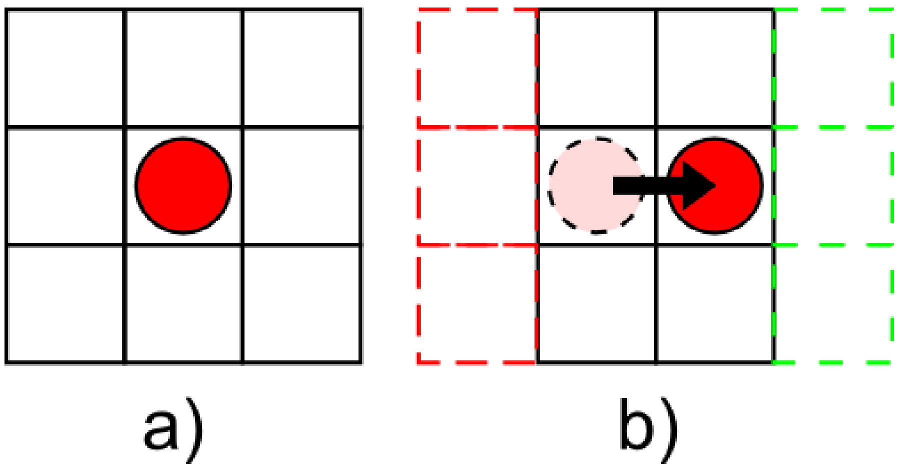


Figure 2.1: In this figure, chunks are represented as rectangles and the viewer as a red dot. a) the viewer stands still - nothing changes. b) when the viewer moves, the chunks behind him (red) are unloaded and the ones in front of him (green) are loaded from the disk.

The LOD-system in Unity also handles terrain-objects which can be used to represent chunks, so vertex-based LOD is already handled by Unity itself [12].

2.6 Unity Terrain-Object

The Unity-terrain-object is a built-in component that enables the user to create heightmap-based landscapes. Each terrain-object can be modified via the Unity-editor using brush-tools or by programming in the scripts. It offers multi-texturing-support and should be used with repeatable(tile-able) textures. The user may also add detail-objects like billboard-grass or rocks to the scene to make it look more realistic. Terrain-objects support trees and offer a LOD-system that will automatically swap distant tree-models with billboards of trees to save computing-power. The user can also define wind - trees and grass will bend in the wind to offer a more immersive scenery.

Because Unity-terrain-objects are heightmap-renderings the user can define the size of the heightmap that should be used. The size of the heightmap is subject to a few restrictions: the heightmap has to be square and the size must be a power of two plus one(e.g. $256 + 1 = 257$). The scaling of the terrain-object in the world can be chosen arbitrarily on each axis. Because of these restrictions it is not possible to handle non-square, non-uniformly distributed measure-points for the height of the terrain. [19]

This object didn't get used in the implementation, because our data set contains non-square patches. More information on the built-in terrain support can be found in the Unity-documentation [19].

CHAPTER 3

Methodology

Because this work is all about testing Unity it got used as an engine and editor along with the provided default tools(like "MonoDevelop-Unity") to create the application. Unity offers 3 scripting-languages to work with: Boo, JavaScript and C#. C# was chosen from the three, because C# is the language closest to hardware and we had the most experience with it. The used test data origins from the Vienna's open government project [20] and has been refined by VrVis. The application was developed and tested on Windows 7 Professional SP1.

In chapter 2, a variety of LOD-algorithms was discussed, but they were not used in this work - except for some suggestions Robert Oates [12] made.

3.1 User Interaction

The user can move the camera around by using the [W], [A], [S], [D]-keys to move horizontally and the [Q] and [E] keys to move the camera vertically. They may use the mouse to look around, but they can only change the yaw of the camera - pitch and roll are fixed. This model of interaction has been chosen for 2 reasons: First, that way it is easier to move the camera around than a totally free camera when looking at the ground. Second, this would avoid the problem of gimbal-locking, but as it turned out this is not a real concern, because Unity uses quaternions to store object rotations.

3.2 OPC-Data-Structure

The data that was used in this work was provided in the OPC-format. OPC stands for Ordered Point Cloud and is a home-made data-format of the VrVis company (see also the paper by Thomas Ortner, Gerhard Paar, Gerd Hesina, Robert F. Tobler and Bernhard Nauschnegg [11]). It is designed to store models of terrain-patches(so called "chunks") and defines a folder-structure as well as the structure of the files containing the terrain-data.

The following tree describes the folder-hierarchy of OPC. In this tree, words in square brackets are variables, e.g. [X] denotes the x-coordinate of the chunk/super-chunk.

- (folder) [DataName]_[X]_[Y].flt
  - (folder)original
    - (file)0.[X].[Y].IntPointChunk.aard
    - (file)0.[X].[Y].PointChunk.aard
    - (file)0.[X].[Y].PointChunk.aari
    - (file)meta.original
  - (folder)subsample
    - (file)LVL_[LOD]_([X].[Y] [Layout])_C.PointChunk.aard
    - (file)LVL_[LOD]_([X].[Y] [Layout])_C.PointChunk.aari
    - (file)cache.subsample
  - (folder)textures

The first folder in the hierarchy represents the super-chunk. Super-chunks represent an additional grouping of terrain-patches: They contain all chunks in a 2560 by 2560 region. Each super-chunk has an x- and an y-coordinate that defines their position. This position is not in world-units, but in super-chunk-steps. In a 2D integer coordinate system, every point represents a super-chunk-position. Super-chunks contain multiple chunks in various LOD (Level Of Detail)-levels. Level 0 is located in the "original" folder. There are three types of files for each chunk in this folder: the original dataset (*.PointChunk.aard), the interpolated dataset (*.IntPointChunk.aard) and a file containing (to us) unimportant meta-informations (*.PointChunk.aari). One super-chunk always contains 5 by 5 level 0 chunks, they might not be equal in size. Additionally, each super-chunk has one meta file (meta.original) that contains useful information like the offsets of the chunks to the origin of the world.

Level 1 chunks and higher are located in the "subsample" folder. Just 2 files describe one chunk: the *.aard and *.aari-file. The *.aari-files got ignored - they only contain some meta-scene data. The *.aard-files contain every information that is needed to display this chunk - it is not a diff-file of the LOD-steps. This has some advantages and disadvantages: On the pro-side, no additional computations are needed to render a single chunk, because all informations are present in the file. On the con-side, this uses more disc-space, because vertices are stored multiple times. Also some algorithms like tessellation require a diff-representation of the LOD-steps - which has to be computed by loading multiple levels of the chunk's detail levels.

Level of detail is constructed as follows: Level 0 has 5x5 chunks per super-chunk that are not compatible with Level 1 - Level 1 has 4x4 chunks that are not of equal size(can be one layout of "LayoutFull", "LayoutHorizontal", "LayoutVertical" or "LayoutSingle"). Level 2 has 2x2 chunks - each chunk has the size of 2x2 Level 1 chunks. Level 3 is just a single chunk representing the hole super-chunk. Super-chunks are of equal size: 2560x2560, so it is possible to swap the whole Level 0 chunks of a super-chunk with the Level 1 ones.

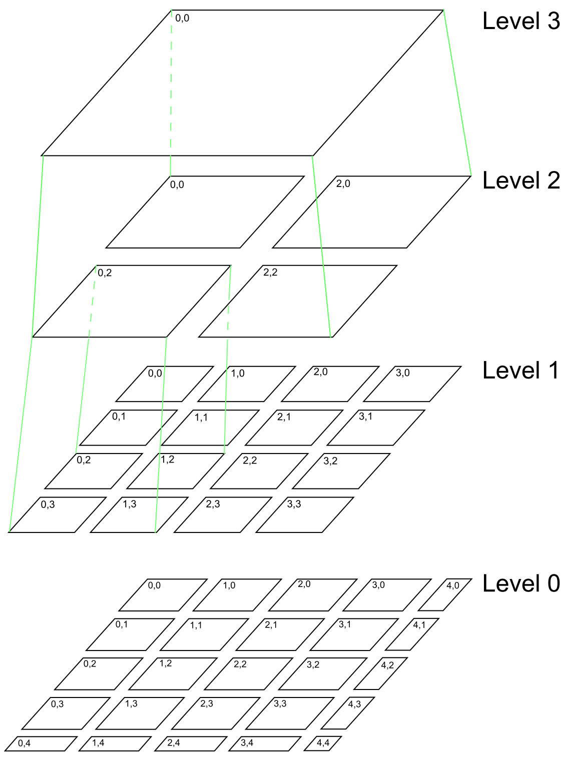


Figure 3.1: Structure of a super-chunk and all of it's LOD-levels

OPC-chunk-files(*.aard) must contain at least 2 datasets: the positions and the index. The organization is the default approach for indexed data: every unique combination of vertex-data is saved under one index. The index is implicitly defined by the position of the vertex data in its list. The index list contains index triples: each one representing an individual triangle. Additionally, it may also contain UV-coordinates, normals and the texture itself in the DXT1-texture format as a blob. The whole file contains binary data: positions and normals are 3 float-variables, 4 bytes each. Indices are integers, 4 bytes each and UV-coordinates are 2 floats, 4 bytes each.

Every OPC-file has a header and a body. The header contains all keywords in UTF-8 encoding followed by the position of the data in the body. The body contains pairs of keywords and blobs: first the keyword in UTF-8 encoding followed by the above mentioned fitting data-set.

3.3 Level of Detail

A simple approach has been taken to solve the LOD(Level Of Detail)-issue. A LOD-graph was used where each node represents one chunk(terrain patch) at a particular LOD-level - with some exceptions. The structure of the LOD-graph is almost a quad-tree. Unfortunately, level 0 patches of the OPC-data-format cannot be grouped together so that they reassemble one level 1 chunk - the smallest grouping that could be found was the whole super-chunk. That means that level 0 can only be represented as a single node per super-chunk. Because we directly load the OPC-data from disc during runtime, we have to represent all 25 Level-0 chunks as a single LOD-node. So in our implementation Level 0 nodes have no children, but 16 parents - every node of the same super-chunk on level 1 is a parent for the Level 0-node. Level 1 nodes have the level 0 node as child - 4 nodes of level 1 have the same level 2 node as parent. There are 4 times 4 by 4 clusters that each belong to a different level 2 node. There are only 4 nodes on level 2 for each super-chunk, so they have 4 children and one parent each: the only level 3 node. The level 3 node has 4 children and no parents. All nodes have 4 connections to their adjacent neighbours on the same LOD-level. Figure 3.2 shows the above described structure of the graph for a single super-chunk.

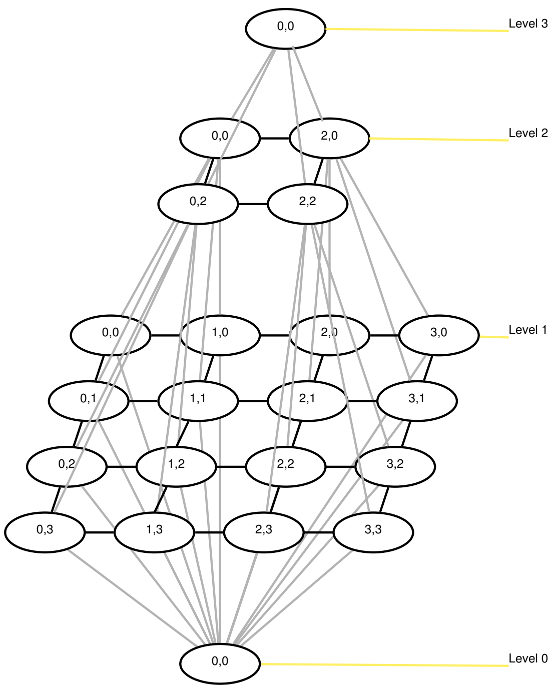


Figure 3.2: structure of a super-chunk's LOD-graph representation

The graph does not represent the whole dataset - it is constructed for the visible part of the world and nodes that are not visible anymore will be removed. The chunk-manager component has a reference to the current node that resides at the middle of the world(at 0,0). When the viewer moves from the middle chunk to an adjacent one, the middle of world changes to the chunk the player moved to. This causes an update of the LOD-Graph:

1. Add LOD-nodes of a super-chunk that are missing, but would be inside of the rendered scene.
2. Update the connections between the LOD-nodes.
3. Remove all LOD-nodes that are outside of the rendered scene.

4. Activate/deactivate LOD-nodes based on their distance to the central node in the graph using a user-defined transfer-function.

5. Tell the chunk-manager to set all chunks that correspond to activated LOD-nodes to be visible in the scene and hide all chunks that don't have an activated node.

While the world is adjusting, the viewer will experience popping artefacts as chunks are being replaced by chunks of another LOD-level. To keep the world center at (0,0), the whole world(including the viewer) will be shifted, so that the new middle actually is located at (0,0). As discussed in section 2.5 this is a necessary step, because Unity uses float-precision for all transformation-matrices. Float becomes numerically unstable at the second decimal place for values of 20,000 or higher. As a single super-chunk has a size of 2560x2560 and we display multiple at the same time, such values can be reached easily.

The viewer will also note some holes in the ground. These artefacts occur at the borders of a chunks with different LOD-levels. Because of the more detailed chunk's higher resolution, the viewer might be able to look through the ground. A possible solution for this problem could be a curtain(see section 2.5) - a mesh that falls like a curtain from the edge of the chunk downwards and blocks the view through the world.

3.4 Chunk Loading

Because of the data size, it is not feasible to load everything at the start of the application - the computer would run out of memory. So we only load chunks in a certain distance from the viewer - additionally, we also keep the last x(selectable by the user) loaded chunks in memory. This speeds up the rendering of previously loaded chunks.

The actual loading is handled asynchronously by a second thread. The main thread requests chunks from the loader-component - this component will then open the specified file, scan it for keywords and read the data. The loaded data will be stored as primitive data types - ready for the main thread to collect. The main thread then picks up the loaded chunk-data and creates the Unity objects necessary to display the chunk - the mesh, the texture and the game-object. This cannot be done by the loading-thread because all Unity-related operations(like creating a mesh) must happen within the main-thread, otherwise an exception is thrown. The only way to load Unity-objects asynchronously is by creating a second scene in Unity - this scene can load objects for you in a second thread [2], but this is slower than the approach we took.

CHAPTER 4

Implementation

There are three main components in this application: ChunkLoader, LOD and ChunkManager. The ChunkLoader loads chunks from files or byte-arrays - it's also responsible for reading the meta-files. The LOD-component includes all classes and interfaces used for the level of detail-graph. The ChunkManager uses the other two components to display the correct chunks.

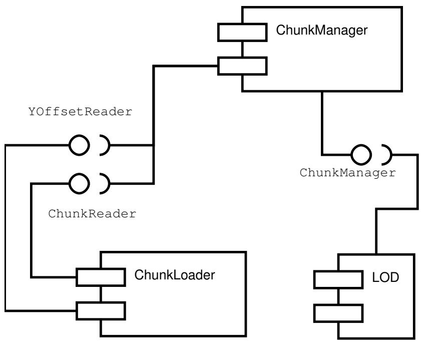


Figure 4.1: Components of the application

4.1 ChunkLoader

The ChunkLoader provides an important interface for loading chunks: the ChunkReader. This interface offers methods for loading a chunk from a file or a byte-array. It is also possible to get just the raw data from the component. The default implementation of the interface is a class called OPCChunkReader. This class uses a BinaryDataSearcher for reading the OPC-file. The BinaryDataSearcher reads a file/byte-array and looks for binary keywords within the data. The OPCChunkReader registers FindingObservers at the BinaryDataSearcher that handle any occurrences of the specified binary keyword. For example, the OPCChunkReader registers a

Vec3Reader (which implements FindingObserver) with the UTF8-binary of the string “Positions” as keyword at the BinaryDataSearcher.

When the BinaryDataSearcher finds a keyword it notifies the observer that has been registered for this keyword. The observer gets the byte-stream at the first position after the keyword. Now the observer can read as many bytes as it wishes - after it is done the BinaryDataSearcher continues the search where the observer stopped. As a stream can only be read forward, only one observer can be registered for one keyword, but observers can observe multiple findings. When the BinaryDataSeracher has finished the search, it notifies all registered FindingObservers that the search has ended, so they can reset themselves.

All implementations of the FindingObserver also extend the AbstractReader. Extensions of this abstract class should read data of the generic type T from the stream and notify all DataReceivers when they are done reading. There are FindingObservers implemented for the positions and normals (Vec3Reader), the uv-coordinates (Vec2Reader), the indices (IntReader) and the texure (DDSReader). All these readers must maintain a set of already found keys, because the key exists twice in an OPC-file: in the file’s header and in the body. So the reader must ignore the first finding and read only the second one (the one in the body). Because four attributes of the mesh are arrays (namely the positions, normals, vu-coordinates and indices) another abstraction-layer got introduced: the PreNumberedBinaryReader with the generic type T. This class assumes that there is a 4-byte integer after the keyword that defines the number of elements to read. So for example, if one had a mesh with 4 points, the integer after the key would be 4. Subsequently the data begins, for the positions this would mean that there were 3 times 4-byte floating point numbers for each vertex (so 12 float-values in our little example). A PreNumberedBinaryReader can read arrays of data only, so it extends the AbstractReader with its generic type T set to T[].

Textures are saved as DXT1 in the file, so the RawDDSReader directly extends the AbstractReader with the generic parameter T set to TextureDescriptor. It reads the DDS-file that is embedded in the OPC-file: it has its own header (128 bytes). First, it checks whether the file is sane (sanity byte @ position 5) then it reads the height and the width of texture from the header. Finally, it reads the body of the texture and saves it as a byte array. The DDSReader has a reference to a RawDDSReader - it lets the raw-reader do its job and then converts the TextureDescriptor into a Texture2D for instant use. This class is not used in the actual program, because chunk loading is asynchronous in the application, but a Texture2D can only be created in the main-thread.

The DataReceiver-implementations get the data of the BinaryReaders, correct them and set them in the OPCChunkReader. For example, the OPCVec3Receiver swaps the y- and z-coordinates of each vector, because in Unity “up” is positive y, but in the OPC-data “up” is positive z. Another correction is made in the OPCIntReceiver: the indices of the mesh are the wrong way around for Unity - it would render all vertices upside-down. So it swaps the last two indices to solve this problem.

After the BinaryDataSearcher has finished its search, the OPCChunkReader has all values set, but does some more correction work before it returns them as a result. Each OPC-chunk has the vertex positions set to an absolute position, so all chunks would form the correct surface. But because of numerical errors described in section 3.3, the viewer will be kept near (0,0), so all chunk-objects need to origin at (0,0) in their local coordinate system. When all chunks have the same relative origin, they can be arranged as desired. So, the OPCChunkReader moves every chunk to (0,0) and saves meta-data like the dimensions and the original position of the chunk. After that last refinement, the chunk is completely loaded (or at least its raw data) and can be returned.

Summarized, the control flow is as follows: The OPCChunkReader starts the search by calling BinaryDataSearcher.searchData(string). The BinaryDataSearcher calls the corresponding BinaryReader for every finding, which loads the data and sends it to all of its DataReceivers. The DataReceivers refine the data a little bit and set it at the correct fields of the OPCChunkReader. After the BinaryDataSearcher is done, all fields that could be loaded successfully are set. After a last refinement the data is returned to the origin of the initial call.

The class YOffsetReader is used for reading the y-offset of a chunk which is used for bringing all chunks to the same vertical level. This offset is equal for each chunk within the same super-chunk, so the loading of the meta-file will be aborted when the first offset has been read. The YOffsetReader uses almost the same interfaces and classes as the OPCChunkReader - the control flow is identical.

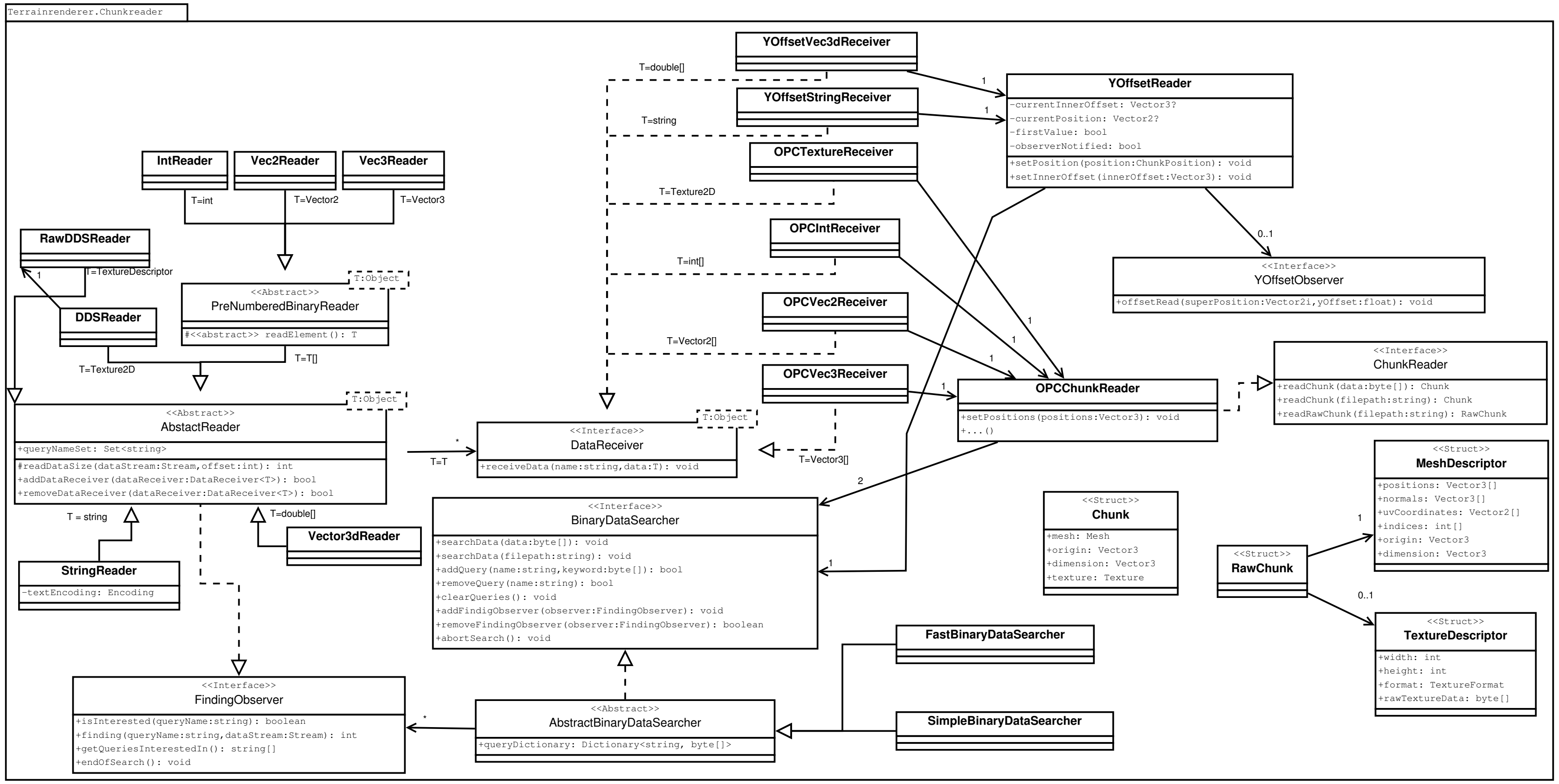

Terrainrenderer.Chunkreader
YOffsetVec3dReceiver
YOffsetStringReceiver
OPCTextureReceiver
OPCIntReceiver
OPCVec2Receiver
OPCVec3Receiver
T=double[]
T=string
T=Texture2D
T=int[]
T=Vector2[]
T=Vector3[]
YOffsetReader
-currentInnerOffset: Vector3?
-currentPosition: Vector2?
-firstValue: bool
-observerNotified: bool
+setPosition(position:ChunkPosition): void
+setInnerOffset(innerOffset:Vector3): void
<<Interface>>
YOffsetObserver
+offsetRead(superPosition:Vector2i,yOffset:float): void
<<Interface>>
ChunkReader
+readChunk(data:byte[]): Chunk
+readChunk(filepath:string): Chunk
+readRawChunk(filepath:string): RawChunk
OPCChunkReader
+setPositions(positions:Vector3): void
+...()
IntReader
Vec2Reader
Vec3Reader
T=int
T=Vector2
T=Vector3
RawDDSReader
T=TextureDescriptor
DDSReader
T=Texture2D
T:Object
<<Abstract>>
PreNumberedBinaryReader
#<<abstract>> readElement(): T
T=T[]
<<Abstract>>
AbstactReader
+queryNameSet: Set<string>
#readDataSize(dataStream:Stream,offset:int): int
+addDataReceiver(dataReceiver:DataReceiver<T>): bool
+removeDataReceiver(dataReceiver:DataReceiver<T>): bool
T=T
<<Interface>>
DataReceiver
+receiveData(name:string,data:T): void
T = string
T=double[]
StringReader
-textEncoding: Encoding
Vector3dReader
<<Interface>>
BinaryDataSearcher
+searchData(data:byte[]): void
+searchData(filepath:string): void
+addQuery(name:string,keyword:byte[]): bool
+removeQuery(name:string): bool
+clearQueries(): void
+addFindigObserver(observer:FindingObserver): void
+removeFindingObserver(observer:FindingObserver): boolean
+abortSearch(): void
<<Interface>>
FindingObserver
+isInterested(queryName:string): boolean
+finding(queryName:string,dataStream:Stream): int
+getQueriesInterestedIn(): string[]
+endOfSearch(): void
<<Abstract>>
AbstractBinaryDataSearcher
+queryDictionary: Dictionary<string, byte[]>
FastBinaryDataSearcher
SimpleBinaryDataSearcher
<<Struct>>
Chunk
+mesh: Mesh
+origin: Vector3
+dimension: Vector3
+texture: Texture
<<Struct>>
RawChunk
<<Struct>>
MeshDescriptor
+positions: Vector3[]
+normals: Vector3[]
+uvCoordinates: Vector2[]
+indices: int[]
+origin: Vector3
+dimension: Vector3
<<Struct>>
TextureDescriptor
+width: int
+height: int
+format: TextureFormat
+rawTextureData: byte[]
1
0..1
2
*

Figure 4.2: Class-diagram of the chunk-loader-component

4.2 LOD

The LOD-component contains the LOD-graph and algorithms to alter the graph. The main interfaces are the LodNode and the LodNodeId. The LodNode represents a node in the LOD-graph - with neighbours (adjacent nodes at the same LOD-level), parents (adjacent nodes at one LOD-level higher) and children (adjacent nodes at one LOD-level lower). For the exact structure of the graph see section 3.3. Each LodNode has a LodNodeId which serves several purposes: First and foremost it serves as a key in hash-tables but can also be used to check if an id is in a set. It can be created without any information about the LodNode, but the position. It also offers a method to create the id of an adjacent node. The second use of the id is to provide the ChunkHandler for the node identified by that id. The ChunkHandler manages the loading/unloading of chunks - it works like a proxy.

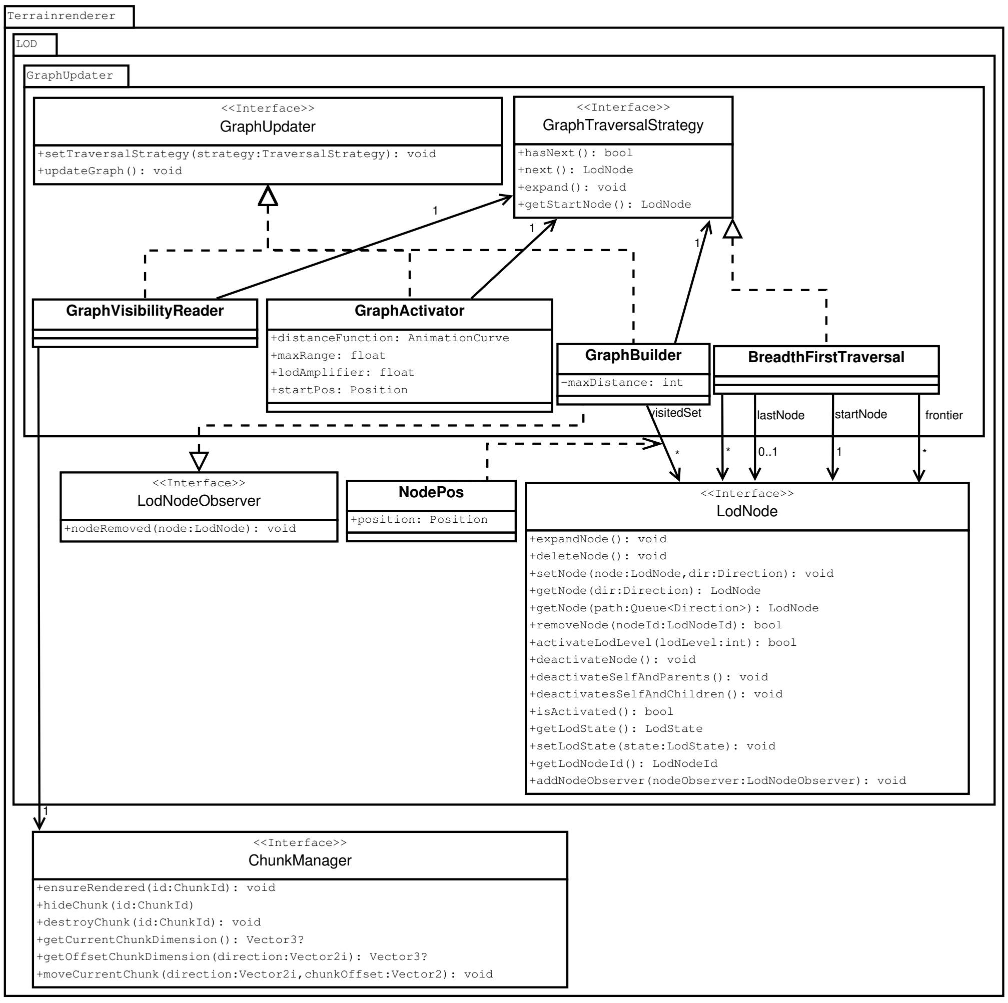
GraphUpdater

Important tasks - such as expanding the graph, computing the active LOD-levels and triggering the loading/rendering of chunks - are done by GraphUpdaters. These classes use an instance of the GraphTraversalStrategy to visit nodes in the graph. Currently, there is one implementation: The BreadthFirstTraversal which performs a breadth-first search on the graph. GraphUpdaters-instances may be reused, but GraphTraversalStrategies have to be re-instantiated for every pass, because they may store data like a visited node set or a frontier-list - they also know the starting-node.

When the center of the world changes, there is a certain order to the different GraphUpdaters. The first GraphUpdater is the GraphBuilder: It stores every node it stumbles upon in its hash-map. When triggered, it traverses through the graph and manipulates the nodes: It uses its hash-map to connect the current node to its neighbours, parents and children. After that, it decides based on its maxDistance-member whether it should expand or remove the node. When a node expands it generates all missing adjacent nodes - these nodes have no neighbours, parents nor children set at this time - it only sets the connection from the current node to the expanded ones. This is necessary, because the traversal-strategy has to be able to find the newly generated nodes.

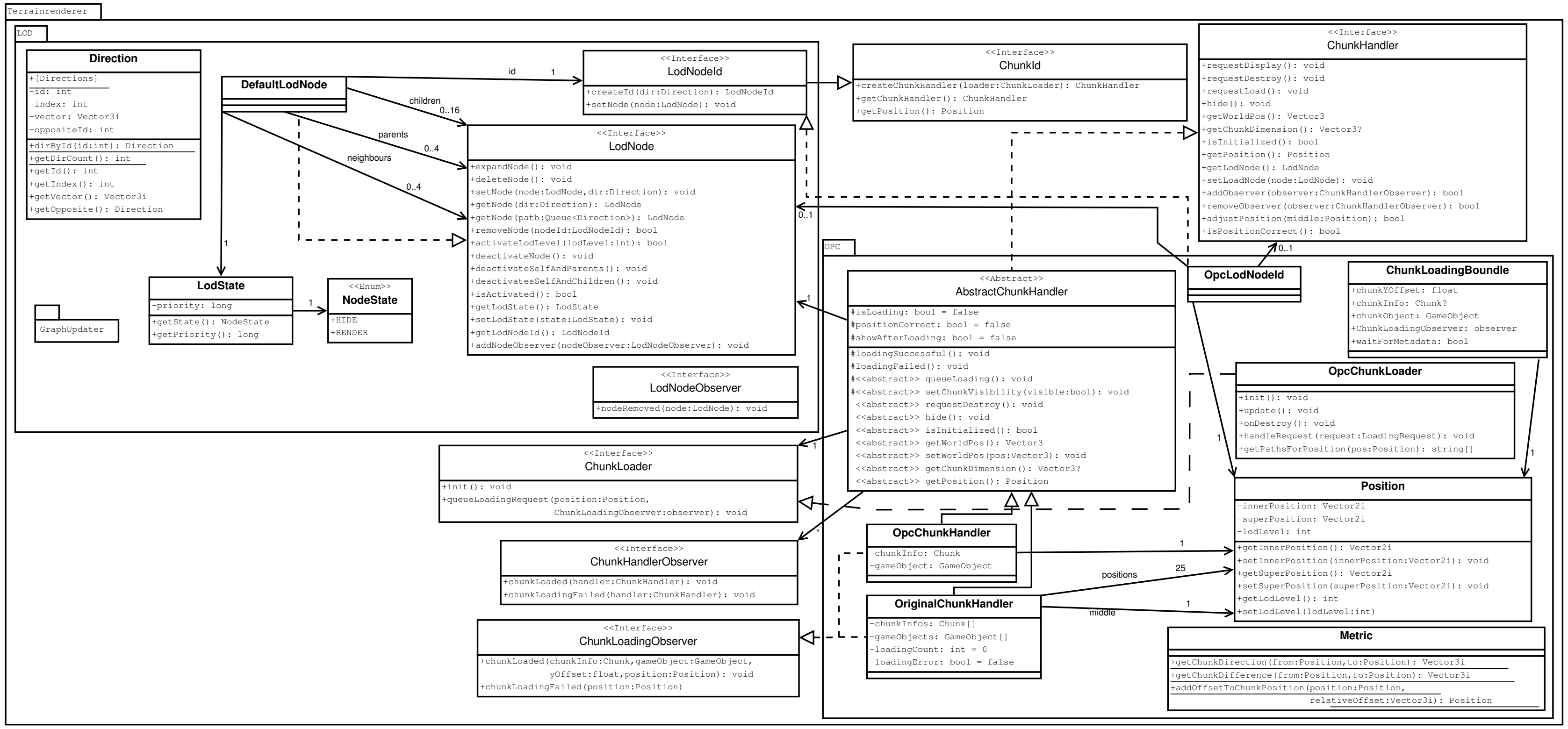
The second GraphUpdater is the GraphActivator. It uses a user defined distance function to activate or deactivate certain LodNodes of the graph. The function can be directly defined in the Unity-Editor, because it is an AnimationCurve. There is a GUI-element in the Unity-Editor that lets users modify this function. The Animation curve is only defined between 0 and 1 on both axes - so the GraphActivator has to stretch both axis so that the x-axis matches the maximum distance in the graph and the y-axis matches the maximum LOD-level available (3 in our case). It traverses through all LodNodes at LOD-level 0 and computes the distance from the center LodNode using the Metric-class. This distance is then used in the distance-function discussed above to get the activated LOD-level. It then calls the activateLodLevel(int)-method of the active LodNode and passes the desired LOD-level. This methods take already activated nodes into account and make sure that only valid constellations of LOD-levels can be activated.

The last GraphUpdater that will be called is the GraphVisibilyReader. This class has the objective to traverse the whole graph, determine if the current node is active or not, calling either ChunkManager.ensureRendered(ChunkId) or ChunkManager.hideChunk(ChunkId) and then setting the node to inactive. It basically tells the ChunkManager which chunks to display and which to hide.



Terrainrenderer
LOD
GraphUpdater
<<Interface>>
GraphUpdater
+setTraversalStrategy(strategy:TraversalStrategy): void
+updateGraph(): void
<<Interface>>
GraphTraversalStrategy
+hasNext(): bool
+next(): LodNode
+expand(): void
+getStartNode(): LodNode
1
1
1
GraphVisibilityReader
GraphActivator
+distanceFunction: AnimationCurve
+maxRange: float
+lodAmplifier: float
+startPos: Position
GraphBuilder
-maxDistance: int
BreadthFirstTraversal
visitedSet
lastNode
startNode
frontier
*
*
0..1
1
*
<<Interface>>
LodNodeObserver
+nodeRemoved(node:LodNode): void
NodePos
+position: Position
<<Interface>>
LodNode
+expandNode(): void
+deleteNode(): void
+setNode(node:LodNode,dir:Direction): void
+getNode(dir:Direction): LodNode
+getNode(path:Queue<Direction>): LodNode
+removeNode(nodeId:LodNodeId): bool
+activateLodLevel(lodLevel:int): bool
+deactivateNode(): void
+deactivateSelfAndParents(): void
+deactivatesSelfAndChildren(): void
+isActivated(): bool
+getLodState(): LodState
+setLodState(state:LodState): void
+getLodNodeId(): LodNodeId
+addNodeObserver(nodeObserver:LodNodeObserver): void
1
<<Interface>>
ChunkManager
+ensureRendered(id:ChunkId): void
+hideChunk(id:ChunkId)
+destroyChunk(id:ChunkId): void
+getCurrentChunkDimension(): Vector3?
+getOffsetChunkDimension(direction:Vector2i): Vector3?
+moveCurrentChunk(direction:Vector2i,chunkOffset:Vector2): void

Figure 4.3: Class-diagram of the GraphUpdater of the LOD-component



Terrainrenderer
LOD
Direction
+[Directions]
-id: int
-index: int
-vector: Vector3i
-oppositeId: int
+dirById(id:int): Direction
+getDirCount(): int
+getId(): int
+getIndex(): int
+getVector(): Vector3i
+getOpposite(): Direction
DefaultLodNode
id
1
children
0..16
parents
0..4
neighbours
0..4
<<Interface>>
LodNodeId
+createId(dir:Direction): LodNodeId
+setNode(node:LodNode): void
<<Interface>>
LodNode
+expandNode(): void
+deleteNode(): void
+setNode(node:LodNode,dir:Direction): void
+getNode(dir:Direction): LodNode
+getNode(path:Queue<Direction>): LodNode
+removeNode(nodeId:LodNodeId): bool
+activateLodLevel(lodLevel:int): bool
+deactivateNode(): void
+deactivateSelfAndParents(): void
+deactivatesSelfAndChildren(): void
+isActivated(): bool
+getLodState(): LodState
+setLodState(state:LodState): void
+getLodNodeId(): LodNodeId
+addNodeObserver(nodeObserver:LodNodeObserver): void
1
LodState
-priority: long
+getState(): NodeState
+getPriority(): long
1
<<Enum>>
NodeState
+HIDE
+RENDER
GraphUpdater
<<Interface>>
LodNodeObserver
+nodeRemoved(node:LodNode): void
<<Interface>>
ChunkId
+createChunkHandler(loader:ChunkLoader): ChunkHandler
+getChunkHandler(): ChunkHandler
+getPosition(): Position
<<Interface>>
ChunkHandler
+requestDisplay(): void
+requestDestroy(): void
+requestLoad(): void
+hide(): void
+getWorldPos(): Vector3
+getChunkDimension(): Vector3?
+isInitialized(): bool
+getPosition(): Position
+getLodNode(): LodNode
+setLodNode(node:LodNode): void
+addObserver(observer:ChunkHandlerObserver): bool
+removeObserver(observer:ChunkHandlerObserver): bool
+adjustPosition(middle:Position): bool
+isPositionCorrect(): bool
0..1
0..1
OPC
<<Abstract>>
AbstractChunkHandler
#isLoading: bool = false
#positionCorrect: bool = false
#showAfterLoading: bool = false
#loadingSuccessful(): void
#loadingFailed(): void
#<<abstract>> queueLoading(): void
#<<abstract>> setChunkVisibility(visible:bool): void
<<abstract>> requestDestroy(): void
<<abstract>> hide(): void
<<abstract>> isInitialized(): bool
<<abstract>> getWorldPos(): Vector3
<<abstract>> setWorldPos(pos:Vector3): void
<<abstract>> getChunkDimension(): Vector3?
<<abstract>> getPosition(): Position
1
1
OpcLodNodeId
ChunkLoadingBoundle
+chunkYOffset: float
+chunkInfo: Chunk?
+chunkObject: GameObject
+ChunkLoadingObserver: observer
+waitForMetadata: bool
OpcChunkLoader
+init(): void
+update(): void
+onDestroy(): void
+handleRequest(request:LoadingRequest): void
+getPathsForPosition(pos:Position): string[]
1
1
<<Interface>>
ChunkLoader
+init(): void
+queueLoadingRequest(position:Position,
ChunkLoadingObserver:observer): void
<<Interface>>
ChunkHandlerObserver
+chunkLoaded(handler:ChunkHandler): void
+chunkLoadingFailed(handler:ChunkHandler): void
<<Interface>>
ChunkLoadingObserver
+chunkLoaded(chunkInfo:Chunk,gameObject:GameObject,
yOffset:float,position:Position): void
+chunkLoadingFailed(position:Position)
OpcChunkHandler
-chunkInfo: Chunk
-gameObject: GameObject
OriginalChunkHandler
-chunkInfos: Chunk[]
-gameObjects: GameObject[]
-loadingCount: int = 0
-loadingError: bool = false
1
positions
25
middle
1
Position
-innerPosition: Vector2i
-superPosition: Vector2i
-lodLevel: int
+getInnerPosition(): Vector2i
+setInnerPosition(innerPosition:Vector2i): void
+getSuperPosition(): Vector2i
+setSuperPosition(superPosition:Vector2i): void
+getLodLevel(): int
+setLodLevel(lodLevel:int)
Metric
+getChunkDirection(from:Position,to:Position): Vector3i
+getChunkDifference(from:Position,to:Position): Vector3i
+addOffsetToChunkPosition(position:Position,
relativeOffset:Vector3i): Position

Figure 4.4: Class-diagram of the LOD-component

4.3 ChunkManager

The ChunkManager is the “heart” of the software - it offers methods to display or hide chunks and takes care that the LOD-graph stays updated. The DefaultChunkManager is also a MonoBehaviour - this enables it to be attached to a Unity-GameObject. When a MonoBehaviour is attached to a GameObject Unity will call some methods by convention (like DefaultChunkManager.Start(), DefaultChunkManager.Update() or DefaultChunkManager.OnDestroy()). The public members can be set by the Unity-editor via a GUI. The distanceFunction is the function used to determine the LOD-level of a LodNode - it can be set via a GUI in the Unity-editor. The maxDistance is the maximum amount of chunks (counted at level 1) that should be displayed at once in each direction around the camera. The maxDestroyListSize dictates how many hidden chunks should be kept in memory at every time. The higher this value is, the more memory will be used up, but if a cached chunk should be displayed again, it is instantly available. startX and startY reassemble the x- and y-coordinate of the super-chunk to start in - the camera will always spawn in the middle of that super-chunk.

The StickyCamera is a MonoBehaviour too - it must be attached to a camera that has at least one parent. This script keeps the camera near (0,0). For it to work it asks the ChunkManager for some information: the size of the current chunk and the size of the surrounding chunks (if necessary). When the camera crosses the border of the current chunk, it will compute the offset needed to turn the new chunk into the current one and triggers that using the ChunkManager. All this happens in the StickyCamera.Update() method which will be called once a tick by Unity itself.

The ChunkManager uses ChunkHandlers to manage the life-cycle of a chunk (load / render / hide / unload). A ChunkHandler manages an arbitrary number of chunks that can be treated as one coherent unit. Chunks at LOD-level 0 are grouped by super-chunks into one instance of OriginalChunkHandler. An OriginalChunkHandler can manage 5x5 chunks of (almost)arbitrary size that form a super-chunk when placed right (see figure 3.1 for the structure of chunks in OPC). All other chunks (LOD-level 1 to 3) will be represented by its own handler: the OpcChunkHandler. This handler only proxies one chunk at a time, so no grouping occurs.

They may manage different amounts of chunks, but they load them the same way: by using a ChunkLoader. The default implementation of this interface is the OpcChunkLoader. It is a Unity-script, because it extends the MonoBehaviour. Chunks can be requested asynchronously - calling objects may register an observer that will be called when the chunk has finished or failed to load. The ChunkLoader itself uses two asynchronous classes: the AsynchChunkLoaderWorker and the YOffsetManager. When a new request comes in, the ChunkLoader first tries to find the corresponding files on the disk. If the LOD-level is 0, the file-name is well defined, but any higher LOD-level has a layout-type in its name. Because there is no reliable way to translate the chunk’s file name from its position, the enumeration-approach has been taken: 4 file-names will be generated (one for each layout) and checked if they exist. The first existing file will be used for loading the chunk.

Subsequently the ChunkLoader requests the chunk and y-offset at the AsynchChunkLoaderWorker and the YOffsetManager. If the YOffsetManager already has a value for the requested superchunk, it will return the offset immediately, otherwise it will attempt to load it. This information is now packed into the ChunkLodingBoundle and saved in a Map (with the position of the chunk as key) for later use. Because the OpcChunkLoader is a MonoBehaviour, it's Update()-method will be called once a tick. This method will first check if the AsynchChunkLoaderWorker has a new chunk. If it has a new one, it retrieves the position of the chunk from the result's user-data. It then looks up the corresponding ChunkLodingBoundle. After using an instance of the ChunkInitializer to convert the RawChunk into an actual Chunk-instance it checks if it's done by looking at the waitForMetadata-flag of the ChunkLodingBoundle. If this is set to true, it will write all acquired data into the ChunkLodingBoundle and continue. Otherwise, it will call the ChunkLoadingObserver to notify it about the success and will transfer the data. It only checks for one loaded chunk per frame, because the conversion from a RawChunk into a Chunk is a heavy operation - too many of them might cause too much lag (framerate drop). After that it will check for any chunks that failed loading and report them to the corresponding observers. Now it's time to look for new meta-data from the YOffsetManager. Similar to the process described above, it will get one piece of meta-data, link it to the ChunkLodingBoundle and check if the loading is done by checking if chunkObject is not null. Observers will be notified if loading is done.

The AsynchChunkLoaderWorker uses a locked request- and two locked result-queues (one for successful and one for failed requests) for accepting requests and returning results. Its main method startLoadingChunks() is executed in a separate thread - it waits for a new request, executes it using a ChunkReader and then waits for the next request in the queue.

The YOffsetManager works similarly, but additionally saves the meta-information in a map for later use. This is performance-friendly, because the meta data doesn't occupy much memory and the meta information is the same for each super-chunk (including all LOD-levels). Because opening files on disk is a time-consuming operation keeping this information in memory is much cheaper.

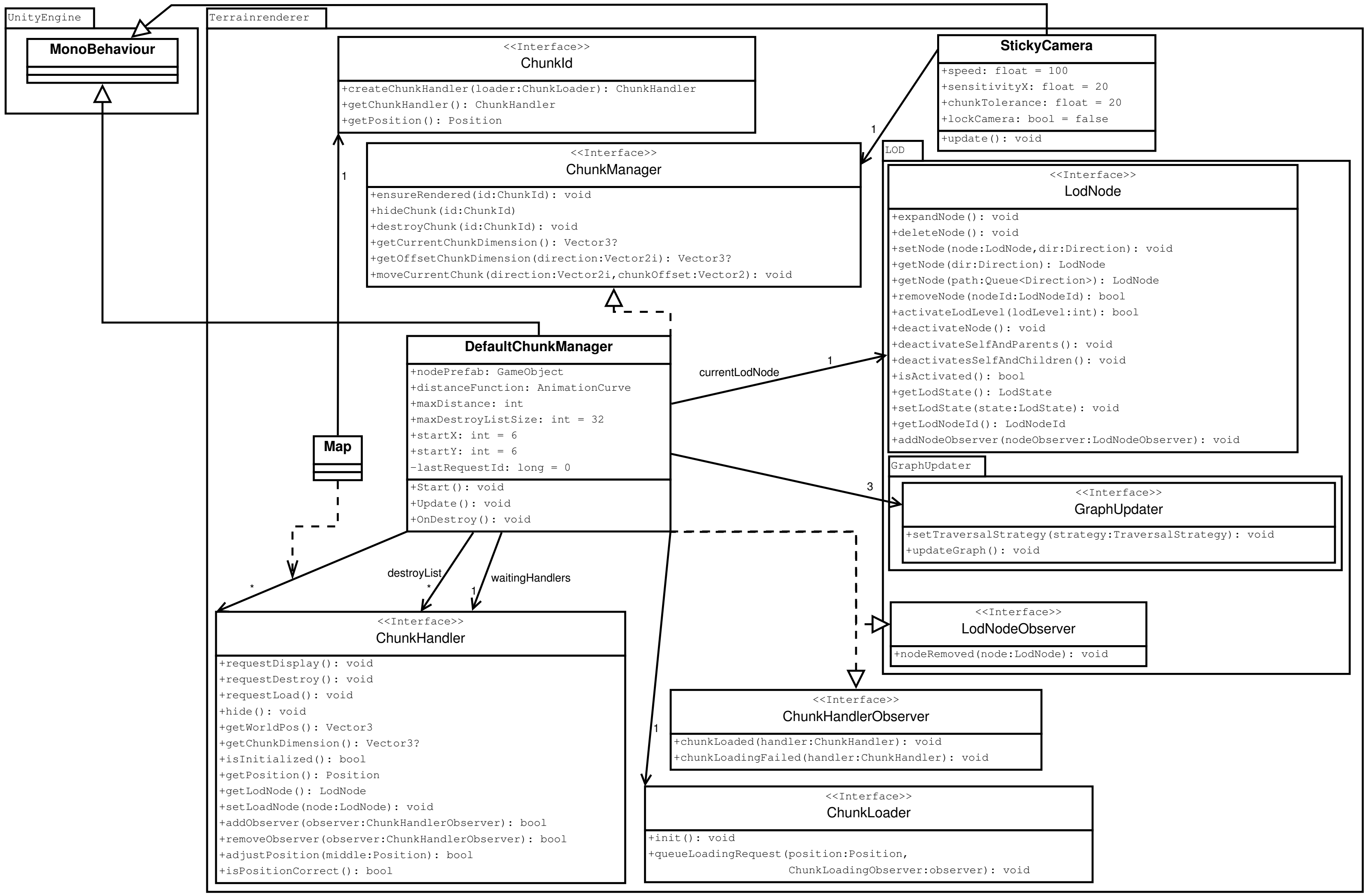
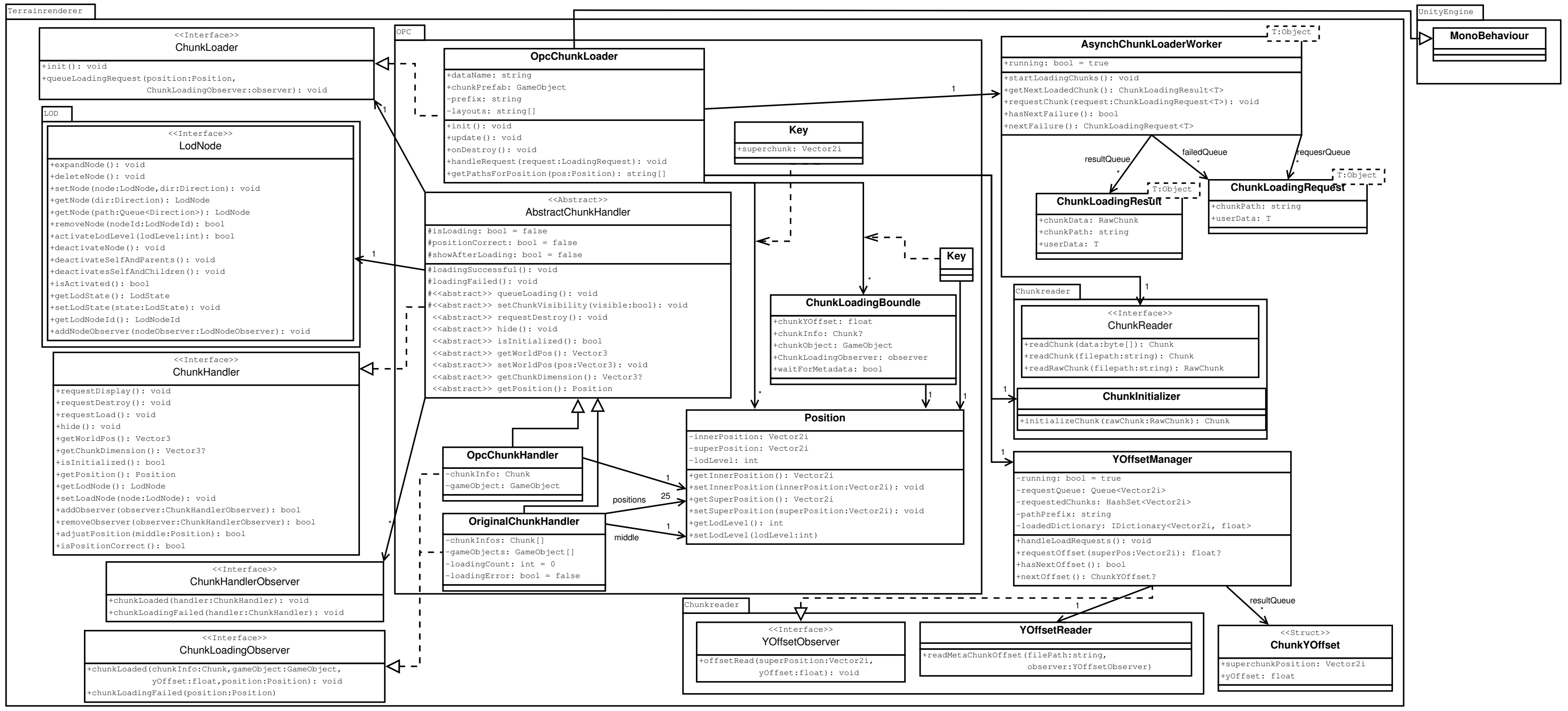

UnityEngine
MonoBehaviour
Terrainrenderer
<<Interface>>
ChunkId
+createChunkHandler(loader:ChunkLoader): ChunkHandler
+getChunkHandler(): ChunkHandler
+getPosition(): Position
StickyCamera
+speed: float = 100
+sensitivityX: float = 20
+chunkTolerance: float = 20
+lockCamera: bool = false
+update(): void
1
<<Interface>>
ChunkManager
+ensureRendered(id:ChunkId): void
+hideChunk(id:ChunkId)
+destroyChunk(id:ChunkId): void
+getCurrentChunkDimension(): Vector3?
+getOffsetChunkDimension(direction:Vector2i): Vector3?
+moveCurrentChunk(direction:Vector2i,chunkOffset:Vector2): void
1
LOD
<<Interface>>
LodNode
+expandNode(): void
+deleteNode(): void
+setNode(node:LodNode,dir:Direction): void
+getNode(dir:Direction): LodNode
+getNode(path:Queue<Direction>): LodNode
+removeNode(nodeId:LodNodeId): bool
+activateLodLevel(lodLevel:int): bool
+deactivateNode(): void
+deactivateSelfAndParents(): void
+deactivatesSelfAndChildren(): void
+isActivated(): bool
+getLodState(): LodState
+setLodState(state:LodState): void
+getLodNodeId(): LodNodeId
+addNodeObserver(nodeObserver:LodNodeObserver): void
DefaultChunkManager
+nodePrefab: GameObject
+distanceFunction: AnimationCurve
+maxDistance: int
+maxDestroyListSize: int = 32
+startX: int = 6
+startY: int = 6
-lastRequestId: long = 0
+Start(): void
+Update(): void
+OnDestroy(): void
currentLodNode
1
Map
GraphUpdater
3
<<Interface>>
GraphUpdater
+setTraversalStrategy(strategy:TraversalStrategy): void
+updateGraph(): void
destroyList
waitingHandlers
*
*
1
<<Interface>>
LodNodeObserver
+nodeRemoved(node:LodNode): void
<<Interface>>
ChunkHandler
+requestDisplay(): void
+requestDestroy(): void
+requestLoad(): void
+hide(): void
+getWorldPos(): Vector3
+getChunkDimension(): Vector3?
+isInitialized(): bool
+getPosition(): Position
+getLodNode(): LodNode
+setLoadNode(node:LodNode): void
+addObserver(observer:ChunkHandlerObserver): bool
+removeObserver(observer:ChunkHandlerObserver): bool
+adjustPosition(middle:Position): bool
+isPositionCorrect(): bool
<<Interface>>
ChunkHandlerObserver
+chunkLoaded(handler:ChunkHandler): void
+chunkLoadingFailed(handler:ChunkHandler): void
1
<<Interface>>
ChunkLoader
+init(): void
+queueLoadingRequest(position:Position,
ChunkLoadingObserver:observer): void

Figure 4.5: Class-diagram of the ChunkManager-component



Terrainrenderer
<<Interface>>
ChunkLoader
+init(): void
+queueLoadingRequest(position:Position, ChunkLoadingObserver:observer): void
LOD
<<Interface>>
LodNode
+expandNode(): void
+deleteNode(): void
+setNode(node:LodNode,dir:Direction): void
+getNode(dir:Direction): LodNode
+getNode(path:Queue<Direction>): LodNode
+removeNode(nodeId:LodNodeId): bool
+activateLodLevel(lodLevel:int): bool
+deactivateNode(): void
+deactivateSelfAndParents(): void
+deactivatesSelfAndChildren(): void
+isActivated(): bool
+getLodState(): LodState
+setLodState(state:LodState): void
+getLodNodeId(): LodNodeId
+addNodeObserver(nodeObserver:LodNodeObserver): void
<<Interface>>
ChunkHandler
+requestDisplay(): void
+requestDestroy(): void
+requestLoad(): void
+hide(): void
+getWorldPos(): Vector3
+getChunkDimension(): Vector3?
+isInitialized(): bool
+getPosition(): Position
+getLodNode(): LodNode
+setLodNode(node:LodNode): void
+addObserver(observer:ChunkHandlerObserver): bool
+removeObserver(observer:ChunkHandlerObserver): bool
+adjustPosition(middle:Position): bool
+isPositionCorrect(): bool
<<Interface>>
ChunkHandlerObserver
+chunkLoaded(handler:ChunkHandler): void
+chunkLoadingFailed(handler:ChunkHandler): void
<<Interface>>
ChunkLoadingObserver
+chunkLoaded(chunkInfo:Chunk,gameObject:GameObject, yOffset:float,position:Position): void
+chunkLoadingFailed(position:Position)
OPC
OpcChunkLoader
+dataName: string
+chunkPrefab: GameObject
-prefix: string
-layouts: string[]
+init(): void
+update(): void
+onDestroy(): void
+handleRequest(request:LoadingRequest): void
+getPathsForPosition(pos:Position): string[]
<<Abstract>>
AbstractChunkHandler
#isLoading: bool = false
#positionCorrect: bool = false
#showAfterLoading: bool = false
#loadingSuccessful(): void
#loadingFailed(): void
#<<abstract>> queueLoading(): void
#<<abstract>> setChunkVisibility(visible:bool): void
<<abstract>> requestDestroy(): void
<<abstract>> hide(): void
<<abstract>> isInitialized(): bool
<<abstract>> getWorldPos(): Vector3
<<abstract>> setWorldPos(pos:Vector3): void
<<abstract>> getChunkDimension(): Vector3?
<<abstract>> getPosition(): Position
OpcChunkHandler
-chunkInfo: Chunk
-gameObject: GameObject
OriginalChunkHandler
-chunkInfos: Chunk[]
-gameObjects: GameObject[]
-loadingCount: int = 0
-loadingError: bool = false
Key
+superchunk: Vector2i
Key
ChunkLoadingBoundle
+chunkYOffset: float
+chunkInfo: Chunk?
+chunkObject: GameObject
+ChunkLoadingObserver: observer
+waitForMetadata: bool
Position
-innerPosition: Vector2i
-superPosition: Vector2i
-lodLevel: int
+getInnerPosition(): Vector2i
+setInnerPosition(innerPosition:Vector2i): void
+getSuperPosition(): Vector2i
+setSuperPosition(superPosition:Vector2i): void
+getLodLevel(): int
+setLodLevel(lodLevel:int)
positions
25
middle
1
*
AsynchChunkLoaderWorker
T:Object
+running: bool = true
+startLoadingChunks(): void
+getNextLoadedChunk(): ChunkLoadingResult<T>
+requestChunk(request:ChunkLoadingRequest<T>): void
+hasNextFailure(): bool
+nextFailure(): ChunkLoadingRequest<T>
resultQueue
failedQueue
requesrQueue
ChunkLoadingResult
T:Object
+chunkData: RawChunk
+chunkPath: string
+userData: T
ChunkLoadingRequest
T:Object
+chunkPath: string
+userData: T
Chunkreader
<<Interface>>
ChunkReader
+readChunk(data:byte[]): Chunk
+readChunk(filepath:string): Chunk
+readRawChunk(filepath:string): RawChunk
ChunkInitializer
+initializeChunk(rawChunk:RawChunk): Chunk
YOffsetManager
-running: bool = true
-requestQueue: Queue<Vector2i>
-requestedChunks: HashSet<Vector2i>
-pathPrefix: string
-loadedDictionary: IDictionary<Vector2i, float>
+handleLoadRequests(): void
+requestOffset(superPos:Vector2i): float?
+hasNextOffset(): bool
+nextOffset(): ChunkYOffset?
resultQueue
Chunkreader
<<Interface>>
YOffsetObserver
+offsetRead(superPosition:Vector2i, yOffset:float): void
YOffsetReader
+readMetaChunkOffset(filePath:string, observer:YOffsetObserver)
<<Struct>>
ChunkYOffset
+superchunkPosition: Vector2i
+yOffset: float
UnityEngine
MonoBehaviour

Figure 4.6: Class-diagram of the ChunkManager-Loading-component

CHAPTER 5

Critical reflection

5.1 Analysis of the Application

The resulting application allows the user to move a camera through an OPC-dataset. When the application changes the LOD-level of chunks, the viewer can briefly see a hole in the ground instead of a chunk and new chunks will pop in as they are loaded. Another artefact a user can observe is that little cracks can occur at the border of two chunks with different LOD-levels. This is because of the different number of polygons - as the vertices don't line up perfectly, the positions may differ and thus create cracks in the landscape. The LOD-selection is based on the distance of the LODNodes in the LOD-graph and not on the actual meshes. This forces an event-based approach for the LOD-system: chunks are only swapped if the graph is re-computed.

A benchmark for the application has also been implemented. This test lets the camera spin around it's own center - showing a 360-degree view of the landscape - and moves it a certain distance in one direction. When the distance is reached, the test stops and the application plots the average, maximal and minimal framerate. The tests were conducted on a TOSHIBA-notebook (PORTEGE R930) with an Intel(R) Core(TM) i7-3540M CPU @ 3.00GHz, with 4GB of DDR3-RAM, an Intel(R) HD Graphics 4000 and a 256GB-SSD running Windows 7 Professional 64-bit. After running a few tests from different starting locations into different directions (but always the same travel-distance) the average framerate was 60fps, the maximum was 246fps and the minimum was 3fps.

The low minimal framerate occurs because of the event-based LOD-system: Every time the viewer crosses the border of a chunk the hole LOD-graph updates. This update alone takes 35.8ms in average on our test-system. The other blocking operation that may also lower the framerate is the chunk initialization: when a raw-chunk gets converted into a chunk. This is the time when meshes and textures get loaded onto the graphics-card. The time for this operation ranges from under 1 millisecond to 46 milliseconds.

5.2 Analysis of Unity3D

During the development, a few obstacles occurred and some things went smoother than expected. In general, Unity works very nicely for the rapid prototyping of games: Its structure with game-objects and components that can be altered even while the application is running makes it perfect to quickly adjust numbers like the movement-speed of the camera. It's also very easy to create and display meshes - no direct interaction with OpenGL or DirectX.

The support for terrain-rendering is not that good: The only object available to render terrain is the Terrain-object. This object uses a heightmap to store its data. This heightmap is an image with a square size of a power of 2 + 1 (9, 17, 33, 65,...). Because the data contained non-square patches, this object was quickly discarded. The only other way to do terrains is by rendering meshes.

While working with Unity, the debugging-feature was used a few times. Most of the time, the debugging went fine, but sometimes the commands "step over", "step into" and "step out" didn't work as expected: sometimes it had the same effect as the "continue"-instruction and sometimes a "step into" behaved like "step over". At some point simply using break-points turned out to be the most reliable way.

Some other times, bugs occurred that were caused by poor in-code documentation. Unity offers the math-class UnityEngine.Mathf which itself offers the static method sig(float). This method returns the sign of any input, but for 0 it returned 1. One would expect the mathematical signum-function which is defined to return 0 for 0. [4] The second problem was caused by c# itself: the modulo-function does not work like the mathematical modulo: it returns the same result for negative numbers as for positive ones, but negative. For example in mathematics -6 mod 5 is 4, but in C# the result is -1.

Another big problem was that Unity runs its editor in the same thread as the program it executes (even the GUI). This is very unfortunate, because if the program gets caught in an infinity-loop, the editor is down too. The only way to fix the situation is to kill Unity's process and restart the whole program. Another bad side effect of this is that the log gets lost too: Unity does not write the log into a log-file, but because the GUI is stuck too it can't display the latest entries that may contain a crucial hint what caused the infinity-loop in the first place.

CHAPTER 6

Summary and future work

The aim of this work was to test whether it is possible to display OPC-data in Unity 3D or not. The short answer to this question is yes. A simple OPC-rendering application has been developed using Unity 3D and using the default IDE “MonoDevelop-Unity”. C# was the language of choice out of the 3 possible scripting languages, because it supports system-close operations like reading files better. Then, an introduction to the methods that were used was given: how the OPC-data structure has been handled, how the LOD-problematic got solved and what kind of interactions the viewer can take while using the application. Additionally, the implementation was described - a close look at all relevant classes has been taken. In the end, the results and benchmarks were discussed. Also, the experience of working with Unity 3D is reflected on - what helped along the way and what was a challenge. In the Appendix you will find some screen-shots of the running application.

Along this way, some challenges occurred: The first one was the OPC-format. It was poorly documented and we had to guess what the data meant. Because of this we ran into several problems: We had to put much more time into reverse-engineering the format than we thought we would have to and it surprised us with interesting design-decisions that one would not expect from such a format (e.g. the terrain height was not normalized - it was offset in several chunks). We have learned that on such occasions, it is better to speak directly to people that actually work with this format, because they know what the quirks are.

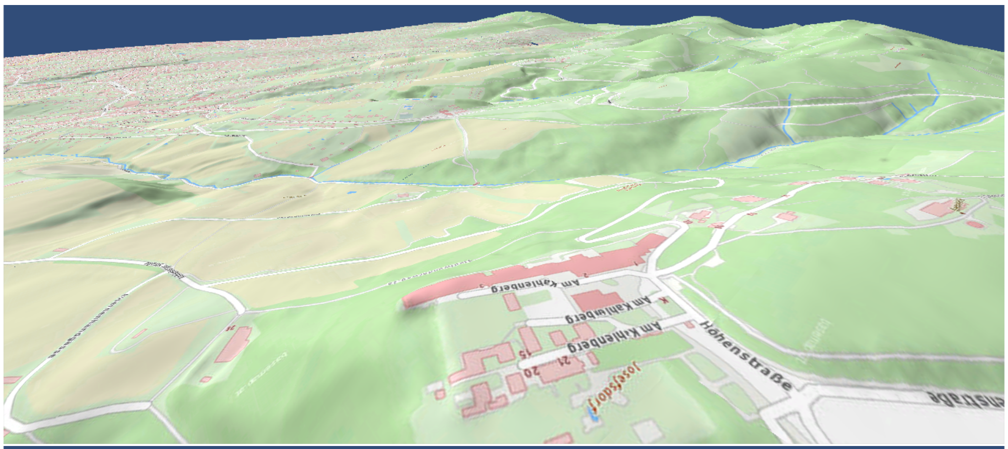
The second big challenge was the LOD-system. We made a slight mistake by over-engineering that component. We wanted it to be data-independent so it could be reused for different projects, but it ended up being too complex to handle, so we had to tear it down and come up with a new, much simpler idea. This was really frustrating and time-consuming. Additionally, this was a goal outside of our original scope creating a lot of problems - we would recommend to limit the amount of nice-to-have goals until an actual working version exists.

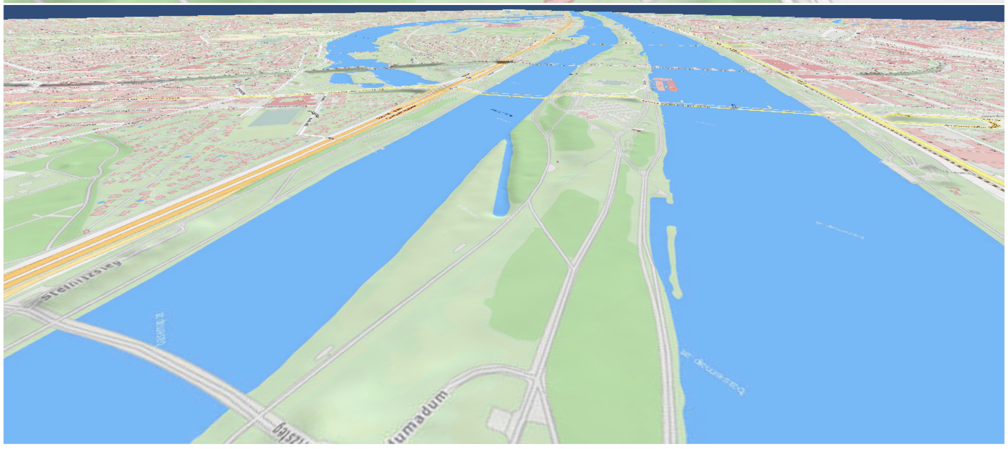
Now that is has been shown that it is possible to create a terrain-renderer using Unity the next step would be to create a better one: using shaders and advanced LOD-techniques the goal should be to create the best possible terrain-renderer using this engine. Another goal would be to do this in a generic fashion: provide interfaces to abstract the work and compile that into a framework to help other developers that want to use Unity for terrain-rendering.

APPENDIX A

Appendix

A.1 Screenshots





Bibliography

[1] Hugues Hoppe Arul Asirvatham. Terrain rendering using GPU-based geometry clipmaps. *GPU Gems 2, M. Pharr and R. Fernando, eds., Addison-Wesley*, March 2005. `http://research.microsoft.com/en-us/um/people/hoppe/proj/gpugcm/`(Accessed: 2014-09-17).

[2] Asynchronous Loading in Unity 3D. `http://docs.unity3d.com/ScriptReference/Application.LoadLevelAdditiveAsync.html`. Acessed: 2014-08-26.

[3] CG-Shader in Unity. `http://docs.unity3d.com/Documentation/Components/SL-ShaderPrograms.html`. Accessed: 2014-03-26.

[4] Definiton of the Signum-Function. `http://en.wikipedia.org/wiki/Sign_function`. Accessed: 2014-09-03.

[5] Embodied GIS HowTo. `http://www.dead-mens-eyes.org/embodied-gis-howto-part-1-loading-archaeological-landscapes-into-unity3d-via-blender/`. Accessed: 2014-04-17.

[6] Hugues Hoppe Frank Losasso. Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Trans. Graphics (SIGGRAPH), 23(3)*, 2004. `http://research.microsoft.com/en-us/um/people/hoppe/proj/geomclipmap/`(Accessed: 2014-09-17).

[7] Import GIS-Data in Unity. `http://answers.unity3d.com/questions/17829/how-can-i-import-gis-data-into-a-unity-project.html?page=1&pageSize=5&sort=votes`. Accessed: 2014-04-17.

[8] Jeff Craighead, HeightmapFromGridFloat. `http://wiki.unity3d.com/index.php?title=HeightmapFromGridFloat`. Accessed: 2014-04-17.

[9] ShaderLab Basics. `http://docs.unity3d.com/Documentation/Components/SL-Reference.html`. Accessed: 2014-03-26.

[10] Filip Strugar. Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD). *`http://www.vertexasylum.com/downloads/cdlod/cdlod_latest.pdf`*, 2010. Accessed: 2014-09-04.

[11] G. Hesina T. Ortner, G. Paar, R. Tobler, and B. Nauschnegg. Towards True Underground Infrastructure Surface Documentation. *Proceedings of CORP 2010:pp. 783-792.*, 2010. http://www.vrvis.at/publications/PB-VRVis-2010-008(Accessed: 2014-09-17).

[12] Terrain Streaming in Unity. http://www.youtube.com/watch?v=VKWvAuTGVrQ&hd=1. Accessed: 2014-03-29.

[13] Terraland. http://www.terraunity.com/product/TerraLand/2/page.aspx. Accessed: 2014-04-17.

[14] Tessellation in Unity. http://docs.unity3d.com/Documentation/Components/SL-SurfaceShaderTessellation.html. Accessed: 2014-03-26.

[15] Thatcher Ulrich. Rendering Massive Terrains using Chunked Level of Detail Control. *http://tulrich.com/geekstuff/sig-notes.pdf*, 2002. Accessed: 2014-03-29.

[16] Unity Game Engine. http://unity3d.com/. Accessed: 2014-03-22.

[17] Unity Licenses. http://unity3d.com/unity/licenses. Accessed: 2014-03-22.

[18] Unity Source Code License. http://blogs.unity3d.com/2009/03/20/why-you-probably-dont-need-a-source-code-license/. Accessed: 2014-03-22.

[19] Unity Terrain-Object. http://docs.unity3d.com/Manual/script-Terrain.html. Accessed: 2014-09-16.

[20] Vienna Open Government. https://open.wien.at/site/datenkatalog/. Accessed: 2014-09-06.

[21] VRVis Forschungs GmbH. http://www.vrvis.at/. Accessed: 2014-03-22.