

# Frontpage for written work/project

<b>ITU student</b> <input checked="" type="checkbox"/> <b>Course</b> <input type="checkbox"/> <b>Thesis</b> <input type="checkbox"/> <b>Project</b> (with project agreement)	<b>EBUSS student</b> <input type="checkbox"/> <b>Course</b> <input type="checkbox"/> <b>Thesis</b> <input type="checkbox"/> <b>Project</b> <input type="checkbox"/> Summerproject <input type="checkbox"/> 4-weeks project <input type="checkbox"/> 12-weeks project <input type="checkbox"/> 16-weeks project
--	---

Title of course, project or thesis:

Game Engines

Course manager/Supervisor(s):

Mark Jason Nelson

Name(s):

1. Emil Erik Hansen

2. Julian Møller

3. Mads Johansen

4. René Bech Korsgaard

5. Steen Nordsmark Pedersen

Birthday and year:

14-06-85

18-03-87

22-05-85

28-05-84

15-02-87

ITU-mail:

emha @

jumo @

madj @

reko @

snop @

Courses with e-portfolio:

Link to e-portfolio: \_\_\_\_\_

# Game Engines

## *Documentation of Engine*

Written by:

Emil Erik Hansen	14-06-85	emha@itu.dk
Julian Møller	18-03-87	jumo@itu.dk
Mads Johansen	22-05-85	madj@itu.dk
René Bech Korsgaard	28-05-84	reko@itu.dk
Steen Nordsmark Pedersen	15-02-87	snop@itu.dk

Group: **Warm Tenderized Fudge**

IT University of Copenhagen  
December 2011

# Contents

<b>Contents</b>	<b>iii</b>
<b>List of Figures</b>	<b>iv</b>
<b>1 Formalities</b>	<b>1</b>
1.1 Documentation Requirements . . . . .	1
1.2 Engine Requirements . . . . .	2
<b>2 Engine Description</b>	<b>4</b>
2.1 Engine Scope . . . . .	4
2.2 Design Rationales . . . . .	4
2.3 Major Features . . . . .	5
<b>3 Implementation Overview</b>	<b>6</b>
3.1 Dynamic Elements . . . . .	6
3.2 Resource Management . . . . .	6
3.3 Rendering . . . . .	7
3.4 Game Loop . . . . .	7
3.5 Input . . . . .	7
3.6 Performance Logging . . . . .	7
<b>4 Examples</b>	<b>8</b>
4.1 Media Manager Usage . . . . .	8
<b>5 Concluding Comments</b>	<b>10</b>

# List of Figures

2.1	Engine Scope: Screenshot from <i>DOTA2</i> . . . . .	5
4.1	MediaManager Usage, Step 1: Initializing containers. . . . .	9
4.2	MediaManager Usage, Step 2: Resources are imported. . . . .	9
4.3	MediaManager Usage, Step 3: Object instantiated. . . . .	9

# Formalities

# 1

The engine was the product of the course *Game Engines*, at the *IT University of Copenhagen*, Fall of 2011.

In this documentation, the engine is described according to the requirements specified in the two following sections.

## 1.1 Documentation Requirements

The following were required to be included in the documentation. The various chapters and sections are structured in such a way that it should be fairly easy to quickly find the needed information, based on the requirements.

- **Engine Scope:** Does it target a specific genre? What range of games do you have in mind that the engine would be suitable for?  
(section 2.1, page 4)
- **Major Features:** What are the main features your engine supports?  
(section 2.3, page 5)
- **Implementation Overview:** Give a high-level explanation of how the engine is structured (perhaps with an architecture diagram). Then, give at least brief explanations of how the major subsystems work, delving into more details for components that are particularly interesting, advanced, or unusual. (section 3, page 6)
- **Design Rationales:** Were some of the design decisions made after considering alternatives, and do you have rationales for why you made the decisions you did? If so, tell us! (section 2.2, page 4)
- **Examples:** Give some examples illustrating interesting features of your engine and how they'd be used. Depending on the features, these can take the form of screenshots, code snippets, flow charts, and/or prose explanation. (section 4, page 8)

## 1.2 Engine Requirements

The following describes the actual requirements of the engine. For each, a reference to the section and page in which it is described is given.

### A Unique/Advanced Element

The feature “Performance Logging” is described in section 3.6, page 7.

- Pick either one area of the engine to add advanced functionality to, or at least one optional feature to add. For example: a user-programmable shader interface, procedural terrain, advanced physics, an audio subsystem, etc.

### Dynamic Elements

Described in section 3.1, page 6.

- *Physics and animation*: What makes sense for your engine and why? Decide on a physics/animation split for your engine, considering its target. Do you want forces, manual animations, parametric curves, something else?
- *Collision detection*: have collider surfaces for your objects, with at least basic collision detection. If you need performance (lots of objects), GJK may be best.
- *Collision response*, if it makes sense for your genre of game: things like objects bouncing off other objects when they collide.
- Be able to *add/remove objects from the game world dynamically*, and update associated data structures.

### Resource Management

Described in section 3.2, page 6.

- *Singleton class* (or several) managing global engine state.
- *Memory-management system*. Design on paper first. Implement at least one kind of custom allocator, e.g. a stack allocator used for per-frame allocations.
- Design a *resource/dependency-management system*, both runtime and offline portion (offline includes things like asset conditioning, and level format).
- Quickest way to get some resources loading: wrap a library like <http://assimp.sourceforge.net>.

## Rendering

Described in section 3.3, page 7.

- *Object-oriented, scenegraph-based rendering*, that recursively renders objects in the scene.
- A *camera component* should be present.
- Use a method of *speeding up rendering* by retaining data on the GPU. For now display lists are okay, but vertex buffers are better (and will be useful later).
- *Lighting manager* (or data structures) storing information on lights in the scene.
- *Material properties* on objects.
- *Textures* on objects.
- *Shaders*: Optional. A user-programmable shader API for your engine could be a nice enhancement. But, everyone should understand how shaders work, even if your engine doesn't use them.

## Game Loop

Described in section 3.4, page 7.

- Support *framerate-independent game-world updates* (e.g. using deltaT timings).
- Implement an *event system* (may be useful to split into several event systems, e.g. an InputManager handling all input events).

## Input

Described in section 3.5, page 7.

- Handle *keyboard* input, with at least a few options. For example: event-based input that queues multiple keypresses, and polling-based input that does something while the key is held down.
- Handle *mouse* (or joystick/controller) input, with the option for nonlinear mapping.
- Connect input to scene in a basic way, e.g. wasd moves an object, mouse moves camera.

# Engine Description

# 2

More of an actual light introduction. Description of the engine and what it can be used for. Covers the ideas and design. See the sections below.

## 2.1 Engine Scope

The engine targets the “Hero RTS<sup>1</sup>”-genre, more commonly known as “MOBA<sup>2</sup>”. This includes games like *Heroes of Newerth* and *League of Legends*, both heavy inspired by the *WarCraft III* modification *Defense of the Ancients*.

To make our engine achieve the requirements of the genre, it was important to have a clear idea of what expected result should look like. As a good reference, see the example screenshot in figure 2.1, page 5.

As can be seen, the basic graphics are fairly simple. There has to be a flat ground-level upon which the player can be controlled around, along with objects he cannot pass. On top of that, there has to be an interactable HUD<sup>3</sup>. The camera will do limited movement, which again will limit the amount of required graphics that needs to be rendered. Furthermore, the avatar of the player has to be able to get from A to B, and be able to pathfind around various obstacles.

This puts the main focus of the engine on effective handling of mouse events, collision-detection and pathfinding.

## 2.2 Design Rationales

Go into more depth here in regards to what actual design decisions were actually made. Was it *this* or *that*? Why was it better than the other?

---

<sup>1</sup>Real-Time Strategy.

<sup>2</sup>Multiplayer Online Battle Arena.

<sup>3</sup>Heads Up Display.





Figure 2.1: Engine Scope: Screenshot from *DOTA2*.

## 2.3 Major Features

To make our game function optimally for the specific game genre, the main - and most powerful - features are the following:

- An effective Mouse Event system.
- A HUD that the mouse can interact with.
- Effective pathfinding.

# Implementation Overview

# 3

More detailed about the structure of the various parts of the engine. The sections below are taken from the requirements of each of the major parts, but might need to be re-structured after that.

## 3.1 Dynamic Elements

**Physics System**

**Collision Detection**

## 3.2 Resource Management

Interesting stuff regarding how Resources in general are being handled - quick introduction, perhaps?

### **Media Manager**

The *Media Manager* is the tool that is to be used in order to import and access media files for use in the game. Currently, it supports importing of .TGA-files as textures and all the common model formats as 3D models, by using *Assimp*<sup>1</sup>. In the future, audio files will be accessed and imported using the media manager as well. As this manager is a singleton, any media imported using it will readily be available anywhere in the code as long you include `<Managers/MediaManager>`.

In order to import a resource, one must first create either a model or a texture image and place it in the Resource folder. A container must then be created for it in the `MediaManager` class (`Texture*` for textures, and `Model*` for models). Then in the `MediaManager.Startup()` function the resource must

---

<sup>1</sup>See *Assimp* documentation for a full list of supported file formats.

be loaded and stored in the aforementioned container using the `LoadTexture()` or `ImportAssimpModel()` respectively. Hereafter the imported asset can be accessed from the `MediaManger` singleton from anywhere within the engine. However, as the scene is actually built in `SceneData.cpp` and the `createGraph()` function, one would usually access in here in order to assign it to an object in the scene. Example of the usage can be seen in section 4.1, page 8.

## Memory Management

Describe the Stack Allocator and Heap Allocators here. It will be interesting and awesome.

## Settings Manager

As every game needs variable settings and options, which should be easy to customize. We decided to keep the data in an external XML-file, so that it is easily changable outside the game. This opens up for the possibility of having a light-weight external tool.

Inside the engine, the XML-file is loaded upon startup and stored as a DOM<sup>2</sup> Tree, which is easily traversable and logically structured. In the same way, the tree can be exported back to the XML-file at any given time, and will also do so upon shutting down the engine.

## 3.3 Rendering

### Camera

### LIGHTNING Manager

:D

## 3.4 Game Loop

### Event System

## 3.5 Input

### Mouse

### Keyboard

## 3.6 Performance Logging

Gogo Gadget Mads!

---

<sup>2</sup>The “Document Object Model”-standard.

# Examples

# 4

Giving examples is one of the requirements of the documentation. It could either be done throughout the document, or simply be here in a chapter for itself. Most likely a better idea, to keep it from the more in-depth technical facts.

## 4.1 Media Manager Usage

The Memory Manager (described in section 3.2, page 6) is used in the following way:

1. Firstly two containers, to hold the player's model and its texture, are created. See figure 4.1, page 9.
2. Then the resources are imported in the `StartUp()` function. See figure 4.2, page 9.
3. And lastly an object is instantiated in the `createGraph()` function and are assigned the player model that we imported earlier. See figure 4.3, page 9.

One might note that the texture is not assigned here. The reason being that it is done automatically when importing the player model as long as the texture is assigned in the model file is called the same as the texture we imported. In this example the name would have to be `player.tga` or `PlayerTexture`.

```
class MediaManager
{
    SINGLETON( MediaManager )

public:
    Model* playerModel;
    Texture* playerTexture;
```

Figure 4.1: MediaManager Usage, Step 1: Initializing containers.

```
void MediaManager::Startup()
{
    ///IMPORTANT: LOAD TEXTURES BEFORE MODELS
    playerTexture = LoadTexture("Resources/playerTexture.tga", "PlayerTexture");
    playerModel = ImportAssimpModel("Resources/PlayerModel.3ds");|
```

Figure 4.2: MediaManager Usage, Step 2: Resources are imported.

```
SceneGraphManager *createGraph()
{
    Object* player = new Object();
    player->model = SINGLETONINSTANCE( MediaManager )->playerModel;
```

Figure 4.3: MediaManager Usage, Step 3: Object instantiated.

## Concluding Comments

5

Might be a good idea with a minor wrap-up at the end.

Also, a better name could be used for this section - but a fitting alternative is eluding me at this point.