IT-UNIVERSITETET I KØBENHAVN

Frontpage for written work/project

ITU student	EBUSS student
[✓] Course	[ ] Course
[ ] Thesis	[ ] Thesis
[ ] Project (with project agreement)	[ ] Project
	[ ] Summerproject
	[ ] 4-weeks project
	[ ] 12-weeks project
	[ ] 16-weeks project

Title of course, project or thesis:

Game Engines

Course manager/Supervisor(s):

Mark Jason Nelson

Name(s):	Birthday and year:	ITU-mail:
1. Emil Erik Hansen	14-06-85	emha@
2. Julian Møller	18-03-87	jumo@
3. Mads Johansen	22-05-85	madj@
4. René Bech Korsgaard	28-05-84	reko@
5. Steen Nordsmark Pedersen	15-02-87	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

List of Figures

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 11)*

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 10.

- 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 associate data structures.

Resource Management

Described in section 3.2, page 8.

- *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 10.

- *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 10.

- 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 10.

- 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`[1]”-genre, more commonly known as “`MOBA`[2]”. 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`[3]. 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?

[1]Real-Time Strategy.
[2]Multiplayer Online Battle Arena.
[3]Heads Up Display.

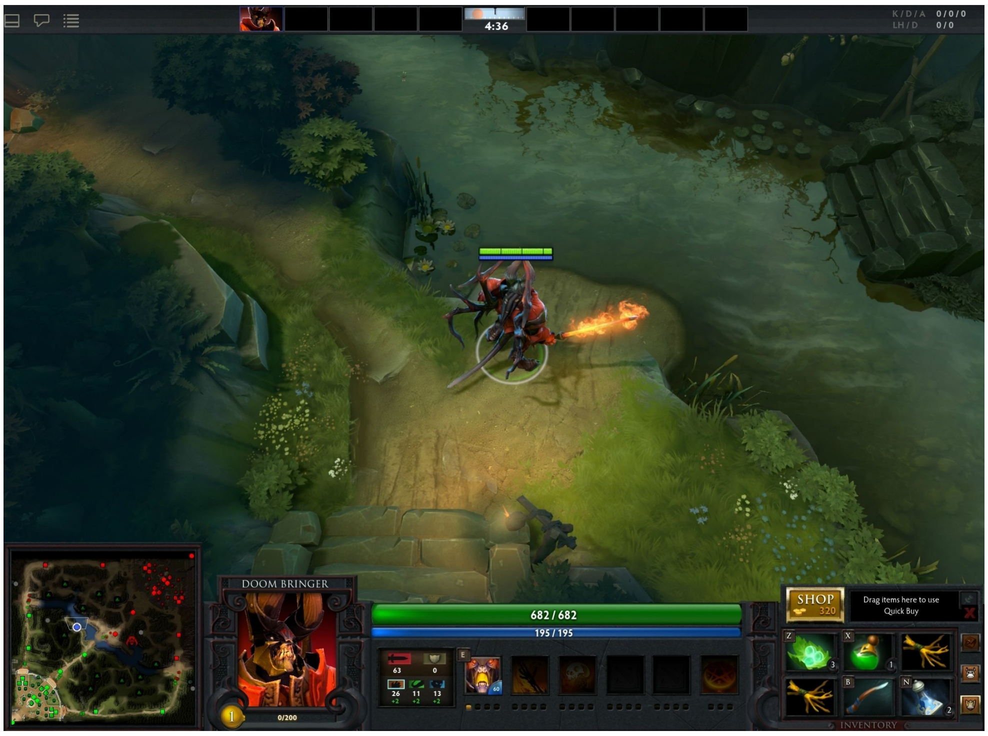


Figure 2.1: Engine Scope: Screenshot from *DOTA2*.

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.

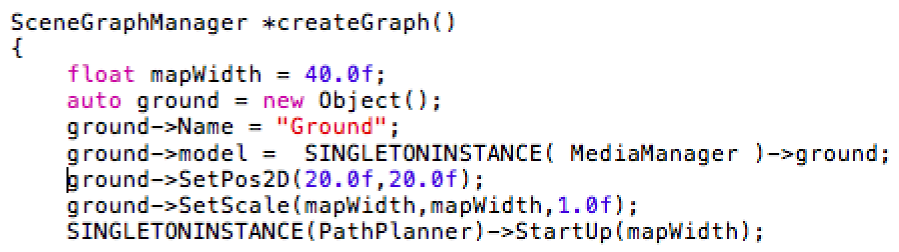
3.1 Dynamic Elements

Physics System

Collision Detection

Path Planner

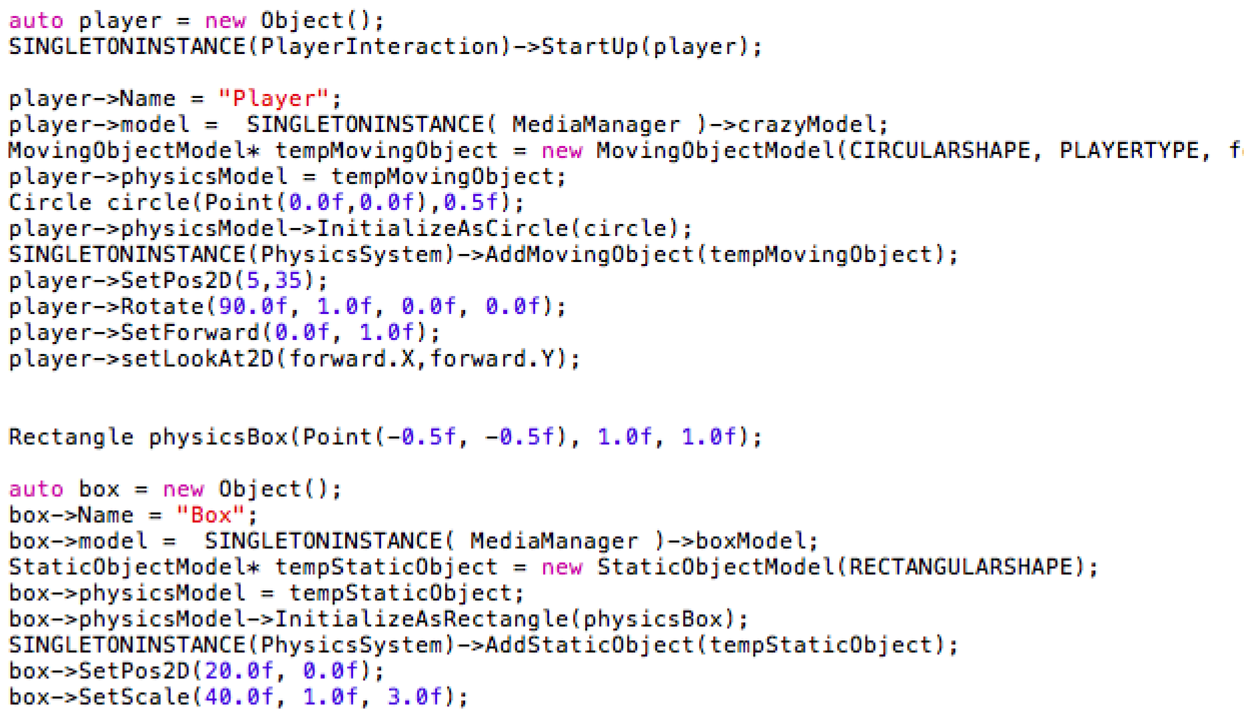
The `PathPlanner` class can be used in conjunction with the Physics System in order to give moving objects in the scene a target destination that they will travel to using the shortest path without traveling into or through other static objects in the scene. Commonly this would be used to give selected units a move-order by using a mouse click event. But before being able to do so, the map must first be initialized. This is shown in figure 3.1, page 6.



```
SceneGraphManager *createGraph()
{
    float mapWidth = 40.0f;
    auto ground = new Object();
    ground->Name = "Ground";
    ground->model =  SINGLETONINSTANCE( MediaManager )->ground;
    ground->SetPos2D(20.0f,20.0f);
    ground->SetScale(mapWidth,mapWidth,1.0f);
    SINGLETONINSTANCE(PathPlanner)->StartUp(mapWidth);
```

Figure 3.1: Path Planner, 1: Initializing the map.

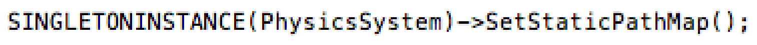
Firstly, the new object is created in the scene, which will be used as the ground and is assigned the ground model, which essentially is a plane width a height and width of one. This is the scaled to a width and height of 40, or whatever one might want, and positioned so that the lower left corner is in $(0, 0)$. Then the width of the map is parsed to to the singleton instance of the PathPlanner.



```
auto player = new Object();
SINGLETONINSTANCE(PlayerInteraction)->StartUp(player);

player->Name = "Player";
player->model =  SINGLETONINSTANCE( MediaManager )->crazyModel;
MovingObjectModel* tempMovingObject = new MovingObjectModel(CIRCULARSHAPE, PLAYERTYPE, f
player->physicsModel = tempMovingObject;
Circle circle(Point(0.0f,0.0f),0.5f);
player->physicsModel->InitializeAsCircle(circle);
SINGLETONINSTANCE(PhysicsSystem)->AddMovingObject(tempMovingObject);
player->SetPos2D(5,35);
player->Rotate(90.0f, 1.0f, 0.0f, 0.0f);
player->SetForward(0.0f, 1.0f);
player->setLookAt2D(forward.X,forward.Y);


Rectangle physicsBox(Point(-0.5f, -0.5f), 1.0f, 1.0f);

auto box = new Object();
box->Name = "Box";
box->model =  SINGLETONINSTANCE( MediaManager )->boxModel;
StaticObjectModel* tempStaticObject = new StaticObjectModel(RECTANGULARSHAPE);
box->physicsModel = tempStaticObject;
box->physicsModel->InitializeAsRectangle(physicsBox);
SINGLETONINSTANCE(PhysicsSystem)->AddStaticObject(tempStaticObject);
box->SetPos2D(20.0f, 0.0f);
box->SetScale(40.0f, 1.0f, 3.0f);
```

Figure 3.2: Path Planner, 2: Creating the two objects.

Secondly, two other objects are added to the scene. These are examples of the two different types of physic objects that are useable. The first object, which is dubbed player is an example of a moving object. It is initialized just like the ground was, but are then added a physics component of a `MovingObjectModel`, which's collider is initialized as a circle. Lastly, the component is added to the list of known moving objects in the `PhysicsSystem`. The exact same thing is done with the box object, but it is just initialized as a `StaticObjectModel` and with a rectangular collider, and is added to the list of known static objects in the `PhysicsSystem`. Note that the player object also is parsed to the singleton instance of `PlayerInteraction`. This is done in order to move it using the `PathPlanner`, when clicking with the mouse on the ground. In figure 3.2, page 7, it is shown how the these two objects are added.



```
SINGLETONINSTANCE(PhysicsSystem)->SetStaticPathMap();
```

Figure 3.3: Path Planner, 3: Static map path.

Then the path planning map is set in the end of the `createGraph()` function. Once this is done, the `PathPlanner` is ready for use. Note that if any objects are added to or removed from the scene dynamically, then the `SetStaticPathMap()` has to be called again. This is shown in figure 3.3, page 8.

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*[1]. 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 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. The Memory Manager (described in section 3.2, page 8 is used in the following way:

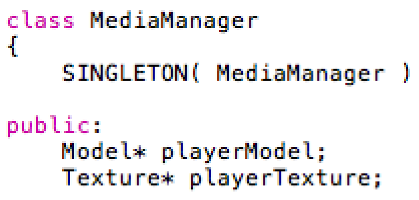
1. Firstly two containers, to hold the player's model and its texture, are created. See figure 3.4, page 9.

2. Then the resources are imported in the `StartUp()` function. See figure 3.5, page 9.

[1]See *Assimp* documentation for a full list of supported file formats.

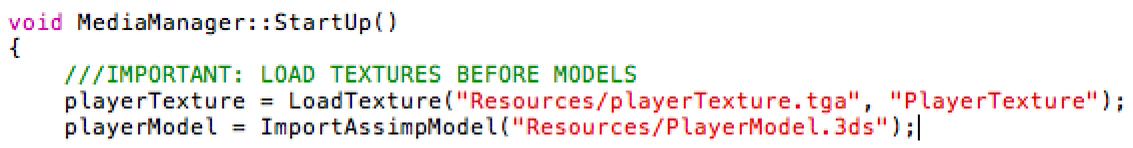
3. And lastly an object is instantiated in the `createGraph()` function and are assigned the player model that we imported earlier. See figure 3.6, page 9.

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`.



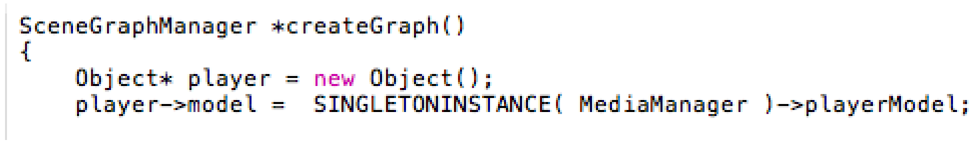
```
class MediaManager
{
    SINGLETON( MediaManager )

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

Figure 3.4: Media Manager, 1: Initializing containers.



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

Figure 3.5: Media Manager, 2: Resources are imported.



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

Figure 3.6: Media Manager, 3: Object instantiated.

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 posibility of having a light-weight external tool.

Inside the engine, the `XML`-file is loaded upon startup and stored as a DOM[2] 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!

[2]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.

It is probably easier simply omitting this section, and describing it when going over the various parts instead. Look at that later.

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.