Developing an Arcade Game

Results of a practical course at the Chair for Computer Graphics and Multimedia
(RWTH Aachen University, Germany)

Stefan Rakel* Patrick Schmidt† Georg Groß‡

Abstract

In this years's practical course *Developing an Arcade Game* we were given the task to develop a game from scratch. Our game called *Shadow Game* implements various sophisticated technologies which are presented briefly in the following sections. These sections are divided into graphics, game logics and editing tools.

Keywords: game programming, shadows, navmesh, deferred shading, ray casting, collision detection

1 General Information

The goal of *Shadow Game* is to reach a chest in the shortest time possible without getting caught by guards. For this the guard's field of view is visualized by the contrast between light and shadow. The experience is enhanced by various power-ups which alter the behavior of the guards. Furthermore you can collect coins to reduce your elapsed time.

2 Graphics

2.1 General Graphics

Shadow Game implements many different graphics technologies which will be discussed in this section. In addition we will take a look at problems that arose during development and how they were overcome.

2.2 Normal and Specular Mapping

All objects in our game heavily utilize normal and specular mapping. Latter simply tells our graphics-engine how much light can be reflected at a particular point as specular light. Former allows us to give the impression of a much more detailed surface than actually present by doing all light calculations as if the surface at any given point is slightly rotated. These techniques together with classic phong-shading result in a visually appealing and realistically looking rendering of our game scene.

2.3 Particle System

A capable particle system for rain and various other effects is also in place. This system is able to spawn particles in a given room of the scene and simulate particle velocities efficiently. It also automatically manages particle lifetimes and variations. All calculations for the particle systems are parallelized in multiple threads to allow a high count of particles. This parallelization is automatically adapting to the count of particles and determines whether multiple threads need to be launched or not.

*stefan.rakel@rwth-aachen.de
†patrick.schmidt1@rwth-aachen.de
‡georg.gross@rwth-aachen.de

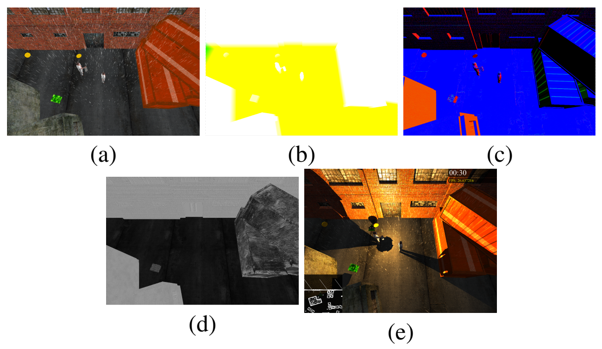

Figure 1: *Contents of the g-buffer: (a)Color, (b)Position, (c)Normals, (d)Specular. And the final result(e)*

2.4 Variance Shadowmaps

All shadows in *Shadow Game* are computed with cubic *variance shadow maps*. The variance shadow maps are not blurred because the game concept demands hard shadows, but this could be easily achieved by simply applying a gaussian blur shader to the maps.

2.5 Deferred Shading

The central idea of *Shadow Game* is to hide in the shadows of light cast by the guards. Because of this the number of shadowcasting lightsources is comparably high. With classic forward rendering techniques many light sources can quickly become a performance issue, because for every object we have to calculate the effects of every light source on it which leads to roughly $L \cdot O$ draw calls with L being the number of light sources and O the number of objects. To overcome this performance bottle neck we implemented deferred shading. Deferred shading renders the scene without light-calculations into a buffer called g-buffer. This g-buffer contains the color, the position, the normal and the specularity of every pixel. Based on this information we then calculate the light only for those pixels that will be presented on the screen. This saves us all lighting calculations for objects occluded by other objects, which brings us to about $L + O$ draw calls.

3 Game Logics

3.1 Architecture

To allow independent work on different parts of the software, we built up an architecture that separates game logics from graphics. For the logics part, we created an entity system that handles all kinds of objects which have an actual meaning to the course of the game. These entities are organized in an elaborate class hierarchy and managed by a central component being responsible for the overall gamestate.

3.2 Collision Detection

We implemented our own 2d collision detection to e.g. check if the player has hit any of his surrounding objects. Every object in our game can have an arbitrary convex bounding polygon and we can ask for intersection of any pair. This is done by a quick bounding circle check followed by applying the separating-axis-theorem to both polygons. In the case of intersection, we also obtain a minimum translation vector that resolves the collision.

3.3 NPC Behavior

To make non-player characters (i.e. guards) act reasonably, each of them uses an internal state machine. Each state defines a specific behavior like patrolling, chasing the player or running to a point of distraction. Transitions from one state to another are triggered by events like establishing/losing visual contact to the player or the effect of power-ups.

3.4 Visibility Ray Casts

A difficult task in our project was how to detect if a guard can see the player. As an additional requirement, we needed to get a rough percentage of how much of the player is visible. We chose to position multiple triggers on the player model and do 3d ray casts from the guards to these triggers. This way, we know how many rays actually hit the player and how many got blocked by scene objects. To speed this up, we are using a three-step approach per ray: 1. check if the trigger is sufficiently close to the guard, 2. do bounding sphere checks with surrounding scene objects, 3. perform a ray-triangle intersection test on the actual geometry of potentially blocking objects. For the last step, we use the well-known algorithm by [Möller and Trumbore 1997]. Some of our models even provide an additional low-resolution mesh wich is used to reduce the number of triangles to be checked. This way, we achieved decent performance even without further acceleration structures. We also do the whole test in parallel.

3.5 Navigation Mesh

Patrolling guards simply walk on a set of fixed waypoints. However, if a guard chases the player he needs the ability to move freely and find a path to the player without hitting any static scene objects. We achieved this by implementing a technique called *Navigation Mesh*. This is a 2d triangle mesh that covers the whole walkable area of a level. Blocking objects, such as containers, are represented as holes in the mesh. Based on the vertices of this mesh, we build up a graph that connects two nodes with an edge if they can be connected by a straight line without leaving the mesh. This is done as preprocessing while loading a level. In the running game, each time we need to find a path, additional source and target nodes as well as the appropriate edges are added to the graph. Since we know that every shortest path can be represented by nodes of the resulting graph, we now use the A* algorithm to obtain the final path.

4 Editing Tools

4.1 Scene representation

Shadow Game uses a custom XML format for an easy and adjustable scene representation. This format allows our game to quickly load various game levels. The XML format is parsed in the startup process and each XML-node type will create a different game object. This set of game objects will be assigned to a level, which is used by our game logic implementation.

As our game uses different logical game objects, this logical separation also exists in our scene format. A scene file can contain several instances of these different logical types, except the player object. This type exists only once as there is only one player. In addition, the scene file holds the path for our navmesh object.

4.2 Scene creation

For scene creation we developed a Python Blender Plugin, which gives us the ability to create and modify our game scene in the 3D editing suite Blender and export this scene directly to a format, which is readable by our scene loader.

The plugin was developed using the standard Blender Python API [ble]. This API gives the exporter the possibility to extend existing Blender functionalities with custom functions. In short, the exporter plugin gives Blender the ability to:

- differ Blender objects to different game objects
- automatic convex bounding box generation (Jarvis march algorithm)
- exporting objects as .obj or own mesh format (with bone animation support)
- converting Blender paths to our waypoint paths and assign to Guard
- automatic NavMesh generation using the Blender game logic implementation

By loading this plugin in the Blender startup process, it will automatic hook in to the default Blender export menu. As it is triggered from this location it will loop through all Blender scene objects and gather the basic object informations (like position, rotation and so on), which are required for every logical game object. Then the exporter will start to decide what kind of logical game object the specific blender object is. From this point, it will start to gather specific information, that are different for each logical game type. In the end all gathered information will be saved in our XML structure.

The Blender Python API stressed us a bit, as it is very version depending and a documentation is rather rare. Therefore, we had to test a lot to gain missing information.

4.3 Own Mesh file

In the development process we came up with the idea to support bone animation for better looking player and guard movements. As the .obj format does not support bone animation we developed an own mesh file based upon XML. Unfortunately the time went short and so we decided to not implement animation support. So the master release still use the obj format but the exporter has the ability to export meshes in our own format.

References

Blender 2.69.10 - API. `http://www.blender.org/documentation/blender_python_api_2_69_10/`.

MÖLLER, T., AND TRUMBORE, B. 1997. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools 2*, 1 (Oct.), 21–28.