Jonathan Moorman
Senior Project
7 June 2011
Dr. Zoë Wood

Third Degree

Introduction

Why a Video Game?

Video games have been driving hardware and graphics development ever since the rise of the first video game consoles. Having a senior project that consists of making a substantial video game is an excellent way to learn about and implement many different types of graphics technologies. Building a video game also provides more experience in all aspects of real world applications development. From the processes of design to team management, the process of building a game will build experience in the many aspects of software engineering. For my senior project, I choose to implement a 3D interactive video game related to the platforming action-adventure genre.

Team and Course Structure

The *Third Degree* game began in Cal Poly's Winter 2010, CPE 476++ - Real-Time 3-D Computer Graphics Software Systems class. The focus of the class was to create a video game based within a 3-D environment, with course requirements to integrate certain graphical technologies into the game. In CPE 476++, Mark Paddon, Chad Williams and Michael Sanchez had the initial idea for a side-scrolling platformer built within a 3-D environment called *Third Degree*. Various people assisted with the project by providing assets and support in the form of music, concept art and models, as well as general story development and voice acting. Tim Biggs, Jon Moorman and Joshua Marcelo joined their programming team to work on the game throughout CPE 476++, as well as continuing on with the project in the Spring 2011 quarter as a senior project. The work on *Third Degree* continued with improvements to graphical technologies while also fleshing out the foundation for the story.

Overview

Genre & Setting

Third Degree is a 3D side-scrolling adventure game. The game takes place in the mind of the main character and the player is continually immersed with story driven game play. Third Degree combines story elements, traditional platforming and interesting game mechanics to provide a unique player experience.

Game Mechanics

The core game mechanic of *Third Degree* is the concept of “mental deterioration” (MD). The status of this “deterioration” state is reflected in the Mental Deterioration Bar, or MD Bar for short. The MD of the player is reflected in many ways in the game. The most apparent affect that MD has is on the game’s environment; initially the environment will reflect the Victorian environment, and will slowly transition to a futuristic setting as the MD increases. Additionally, the higher the MD, the more the distorted the environment will become, furthering the concept of the player’s mental state. As the MD reaches nearly full (equivalent to “death”), the player must *focus* in order to bring down their MD and restore the environment to a playable state. If at any point the player maxes out their MD, they will respawn to the last checkpoint.

Aside from the MD, the player will be fighting against enemies in the twisted environment that the game is set in. Enemies are strategically placed throughout the map, and the enemies’ attacks will increase the player’s MD for each “hit” on the player. For defense, the player is equipped with a melee attack, as well as a simple gun. In addition to the enemies impeding progress in the map, *puzzle objects* are placed throughout to make navigation more difficult. Examples of puzzle objects include trapdoors, swinging platforms, and swinging blades. To pass some of these puzzles, “Fire legs” must be employed. *Fire legs* is the game’s mechanic to allow the player to jump higher than they normally would; the jump height increases with the player’s MD, giving the gamer an incentive to balance their MD appropriately.

Third Degree Story

The game follows the story of a convict kept in confinement who is essentially given a chance at redemption through a special testing program. A recent alien artifact has crash landed on the Earth’s surface, and a panel of scientists are conducting specialized experiments to find out what it does. The convict is one in a line of test subjects given a chance of freedom through experimentation. When the artifact is fitted on the convicts head, he is put in some kind of virtual environment that resembles London in the 1860s. The convict, though determined to gain his freedom, soon feels

the grasp of insanity closing in around him, and the only way out is to either finish the virtual simulation, or die trying.

Project Design Specifics

As a development platform for Third Degree, Microsoft Visual Studio 2008 was used. For our editor UI system we used Nokia's Qt libraries and tools. The Visual Studio plug-in was used to integrate Qt and facilitate the design of the Level Editor. The source code was under version control and the primary language was C++. The repository for our entire project was hosted on Unfuddle.com. All team members were given accounts and access to the repository to commit and update changes to their projects accordingly.

Related Works

While there were many influences for *Third Degree*, the following works both influenced the gameplay as well as helped to provide examples for how to approach certain tasks for components of the *Third Degree*.

Trine

The general mood and feel of the game were greatly influenced by this side-scrolling platformer, Frozenbyte's *Trine*. Visual inspirations, as well as overall feel of the gameplay mechanics such as movement, puzzle object interaction and elements of the combat system helped in making decisions for *Third Degree*. The Figure below shows an in game screenshot for *Trine*.

Fig. 1 – Screenshot for *Trine*

Maya

The transformation tools were modeled after many 3-D graphics software, particularly Autodesk Maya. The figure below shows an example of the transformation tools used in Maya.

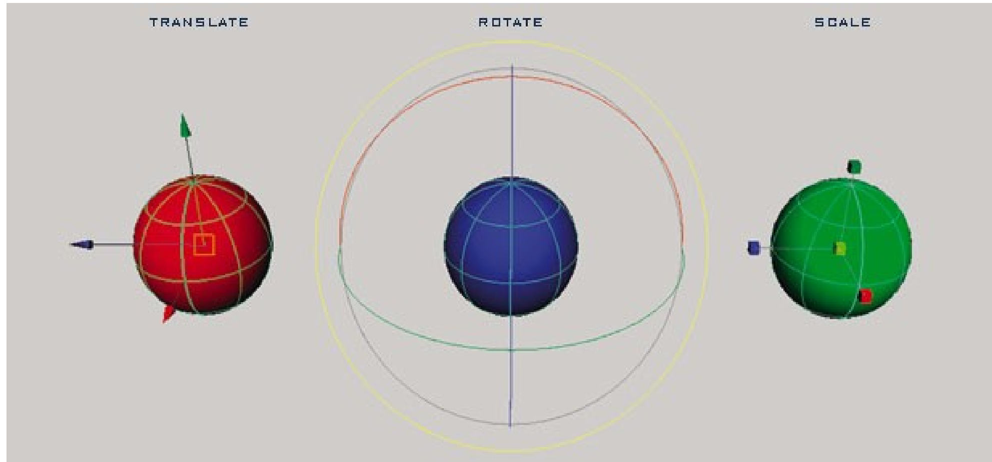


Fig. 2 – Examples of the Translate, Rotate & Scale tools in Autodesk Maya.

Doom 3

The animation in the game utilizes the md5 format used in a number of 3-D games, most notably in Activision's *Doom 3*. The MD5 structure created for *Doom 3* provides a robust and efficient way for representing animation. The figure below shows an example mesh from *Doom 3* modeled using MD5.

Fig. 3 – Example of an MD5 model used in *Doom 3*

Algorithms Overview

Project Technologies

Technology	Authors/People involved
View Frustum Culling	Josh
Skeletal Animations	Josh
Enemy AI	Josh
Player Control/Movement	Josh, Tim
Combat System	Josh, Tim, Mark
Editor - Object Transformations	Josh
Editor - Main Functionality	Chad
Deferred Rendering	Chad, Ryan Schmitt
GLSL Shaders	Chad
Core Engine Optimizations	Chad
Particle System Implementation	Chad
High Level Design	Jon, Mark
Map Loading/Saving	Jon
Physics Engine Integration	Jon, Tim, Mark
Object/Joint System (aka Puzzle Objects)	Jon, Tim
Glow Shader	Jon, Chad
Animated Textures	Michael, Mark
Menu System	Michael
Fire Legs Implementation	Michael, Tim
OBJ Importer	Michael, Chad
Octree	Mark

Sound Design	Mark
Focus	Mark

Algorithms Details

High Level Design

The solution is broken up into three main components. These are the graphics engine, the level editor, and the game logic. This logical separation is done so that both the level editor and the game, which will be running separately, can both use the common rendering code contained in the graphics engine. The graphics engine handles everything related to the drawing of the scene. It determines which objects to draw based on view-frustum culling, applies a variety of shader effects and handles all of particle effects in the scene. The editor makes use of the engine to provide a graphical tool for the construction of levels. It allows a user to edit a level without the tedious guess-and-check process that editing a text file would require and with the flexibility not afforded by building a level in the code. The game logic is completely independent from the editor but uses the same graphics engine to display the current state of the game.

The engine maintains various lists that it iterates over to properly render each scene. Everything in these lists is a GameObject. The GameObject hierarchy shown in Figure 4 is the core representation of the world. This is a very object-oriented approach that yielded a number of benefits. GameObject is the abstract class at the top of the object hierarchy. This means that every object in the world is a GameObject. A GameObject has a position, a rotation, and a scale, as well as an axis-aligned bounding box. All GameObjects also have a draw method which allows the engine to tell each object to draw itself properly. With this information, rendering and view-frustum culling can be done on any GameObject in the scene. The subclasses of GameObject separate the objects into general categories where the drawing code becomes more specialized and additional attributes are added to each object to allow it to perform its functions.

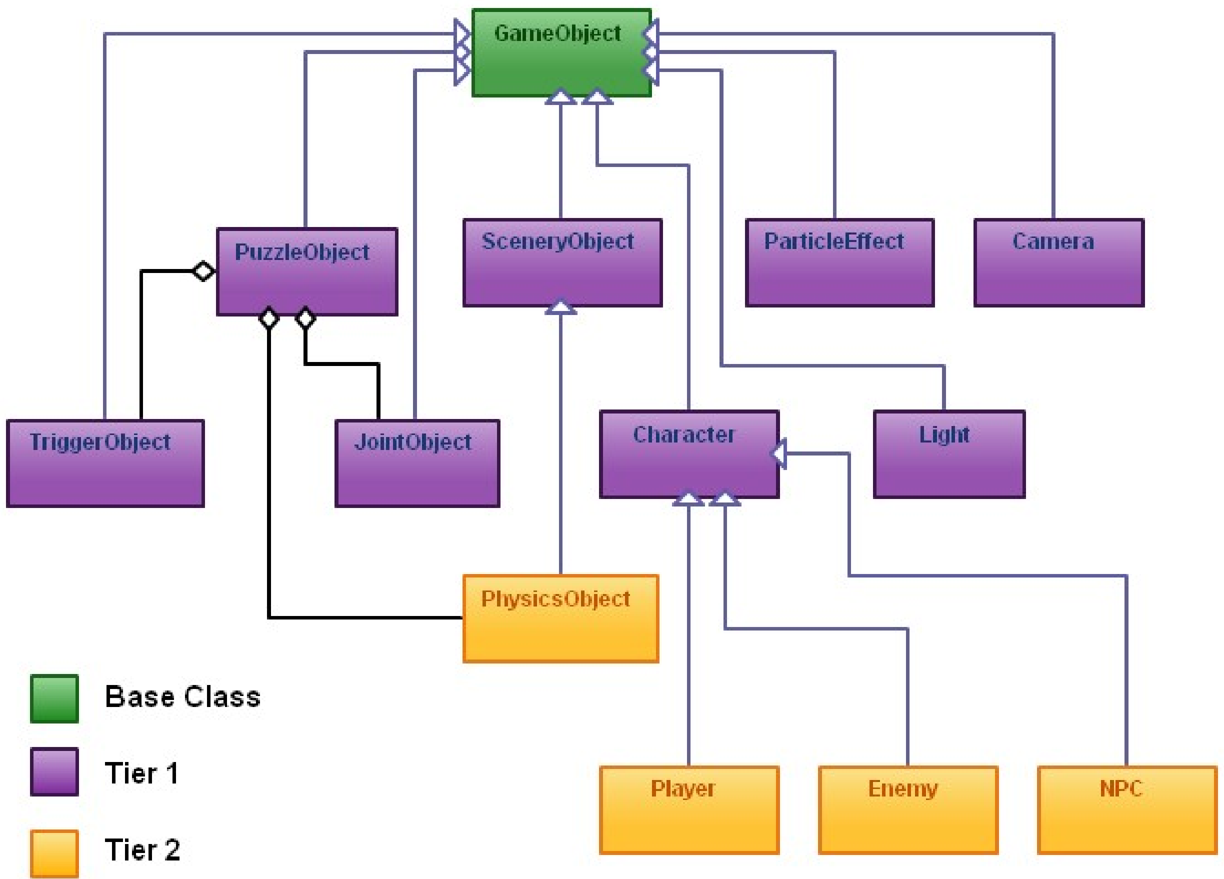


Fig. 4 – GameObject class hierarchy

Physics Engine Integration

This project uses the NVIDIA physics engine, PhysX, to simulate physical interactions with objects in the world. The way that the simulation links itself into a project is through a generic field that is a member of every object in the library. This is the userData field and can hold a pointer to any piece of data. PhysX uses rigid bodies known as actors to simulate gravity, collisions and motion. In the game, there is a special subclass of GameObject called PhysicsObject. A PhysicsObject is an object that in addition to its drawing data, it also contains all of the data required by the PhysX simulation. Every PhysicsObject has an actor associated with it that acts as its physical body in the scene. When drawing itself, a PhysicsObject can query its actor for information about position and rotation. This link is reinforced by having the actors' userData field point to the corresponding PhysicsObject. This is used for things such as collision detection where it is necessary to find which object in the game corresponds to a specific actor. These links allow the physics engine to be seamlessly integrated into the existing object hierarchy and work alongside the game logic.

Glow Shader

Real-time glow was added to the game by applying a post-processing affect to the rendered scene. In addition to providing a diffuse texture, an object that will glow also needs a glow texture. The glow texture is a black and white texture that is black everywhere except for portions of the object which should be sources of glow.

The scene is rendered using a combination of the glow texture and the diffuse texture to effectively mask out all portions of the object which should not be glowing. With only glow sources left in the scene, the next step in the process is to blur the image to soften the glow and make it appear as though light is emanating from the objects. The blur is done by having each pixel take a portion of the color of all surrounding pixels. This is done in a two-pass approach known as a separable convolution. If each pixel were to collect information from all pixels n units away in a single step, the resulting complexity would be n-squared per pixel. By separating the blur into a horizontal blur and a vertical blur, the complexity can be reduced and done in linear time. The image resulting from this process can then be blended with the unaltered scene to add the final glow effect. The glow shader was then integrated into the project by Chad so that it could interact properly with the deferred lighting.

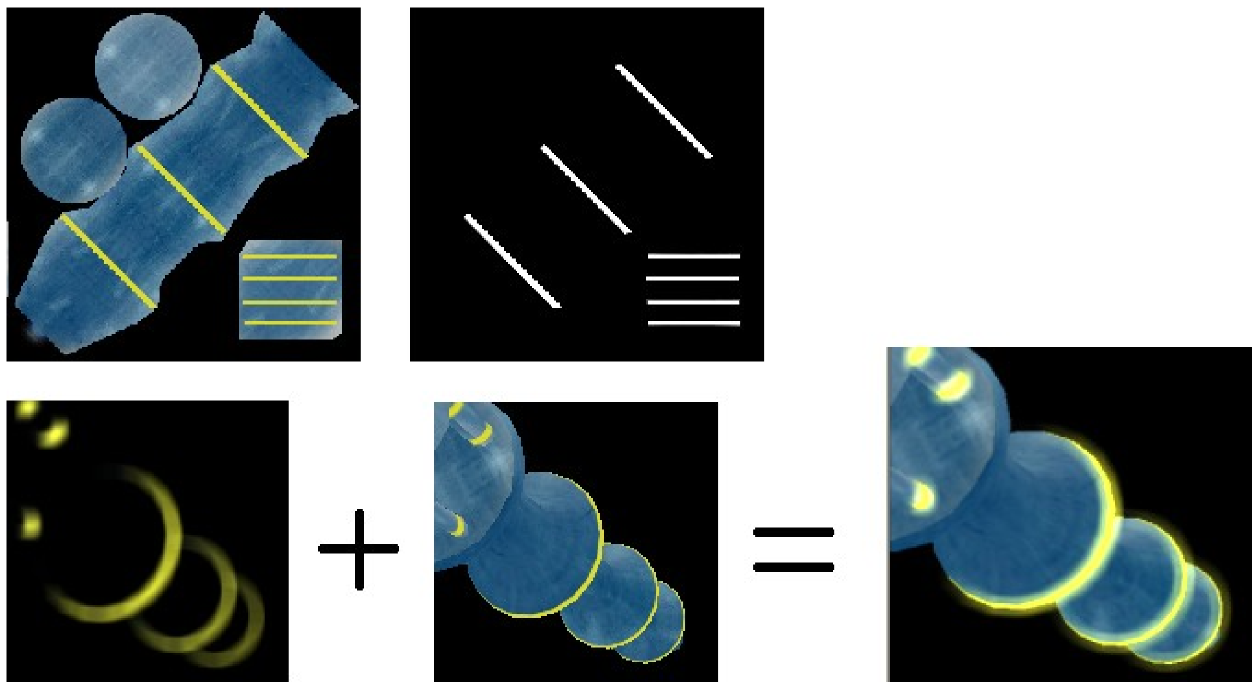
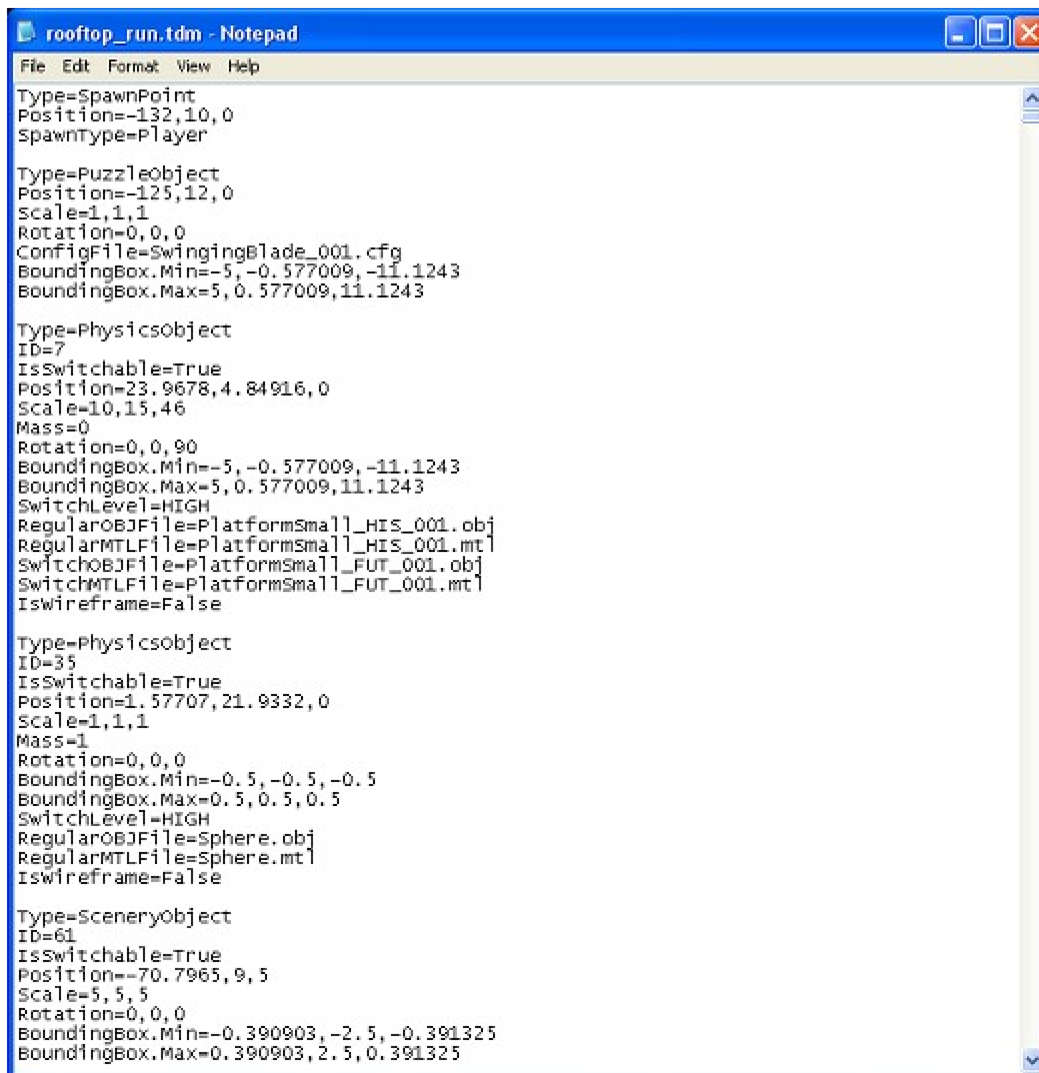


Fig. 6 – Glow Effect

Map Loader

The map loader is what allows the creation of map layouts and scenarios outside of a programming environment. It is also the bridge between the level editor and the game itself. The map loader deals will all of the loading and save of maps to and from

text files. Because the map loader is responsible for both the loading and saving, it can safely assume that it will not be given invalid data. With this assumption in mind, robustness can be sacrificed for speed. Load times are almost always annoying to users and anything that can be done to speed up that process is usually appreciated. An example map file can be seen in Figure 7. To do a load, the map loader reads each map file line-by-line, and creates the appropriate object. The objects are set with the properties specified in the map file and added to a list. This list is then returned to the calling function. To save a map out to a file, the reverse is done. The program goes through the lists of objects and writes each object back to the file with all of its current properties.



rooftop_run.tdm - Notepad

File Edit Format View Help

```
Type=SpawnPoint
Position=-132,10,0
SpawnType=Player

Type=PuzzleObject
Position=-125,12,0
Scale=1,1,1
Rotation=0,0,0
ConfigFile=SwingingBlade_001.cfg
BoundingBox.Min=-5,-0.577009,-11.1243
BoundingBox.Max=5,0.577009,11.1243

Type=PhysicsObject
ID=7
IsSwitchable=True
Position=23.9678,4.84916,0
Scale=10,15,46
Mass=0
Rotation=0,0,90
BoundingBox.Min=-5,-0.577009,-11.1243
BoundingBox.Max=5,0.577009,11.1243
SwitchLevel=HIGH
RegularOBJFile=PlatformSmall_HIS_001.obj
RegularMTLFile=PlatformSmall_HIS_001.mtl
SwitchOBJFile=PlatformSmall_FUT_001.obj
SwitchMTLFile=PlatformSmall_FUT_001.mtl
IsWireframe=False

Type=PhysicsObject
ID=35
IsSwitchable=True
Position=1.57707,21.9332,0
Scale=1,1,1
Mass=1
Rotation=0,0,0
BoundingBox.Min=-0.5,-0.5,-0.5
BoundingBox.Max=0.5,0.5,0.5
SwitchLevel=HIGH
RegularOBJFile=Sphere.obj
RegularMTLFile=Sphere.mtl
IsWireframe=False

Type=SceneryObject
ID=61
IsSwitchable=True
Position=-70.7965,9,5
Scale=5,5,5
Rotation=0,0,0
BoundingBox.Min=-0.390903,-2.5,-0.391325
BoundingBox.Max=0.390903,2.5,0.391325
```

Fig. 7 – Example map file

Results

Playtester Feedback

During testing there was lots of feedback given to help improve the game and determine what was working and what was not. Some of the aspects that people liked in the game was the graphically rich environment, with emphasis on the lighting. People also liked the physics effects of the environment objects and the game actually being a side scroller. Some aspects that people did not like included not being able to see the bullets, excessive movement of the camera, inconsistent frame rates in certain areas of the map and the gun of the player being too small.

Conclusion

The process of making a game with a team of developers was a valuable learning experience. To create a successful project requires proper management, a diverse skill-set amongst developers, and a drive to succeed.

Initially, a large group of 3D modelers and animators was brought onto the project as well as another programming team to work on the AI. Unfortunately, the 3D modelers were somewhat unreliable and we had no way of enforcing any kind of management practices because all of the work was on a volunteer basis. We ended up having to cut most of the team from the project. The real lesson is that when working in a small team with concrete deadlines, outsourcing certain tasks can be dangerous because the work that you expect to be done may never come to fruition.

One thing that was invaluable to the project was the variety of different skills that our development team had. Many of us had taken various software engineering, graphics, artificial intelligence, and algorithms classes. After developing an initial design and architecture(which came far too late in the process to avoid the complete overhaul halfway into the project) we were able to move people around from various tasks to maximize our efficiency.

The most important piece of any software project involving multiple developers is good design. Without a good design that everyone understands, the project can quickly devolve into a complete mess. With a good design, each developer knows what they need to be working on and where in the design that piece of code fits.

Future Work

During the summer, the team plans to continue on with the development of Third Degree. The ultimate goal is to publish a full version of the game on Steam's game store under the Indie game category. The beginning steps to this goal are to have a stable version of the game as well as a well developed story with corresponding gameplay. This includes building an adequate number of levels that will allow the story to proceed at a modest rate while at the same time maintaining the gameplay elements that were initially planned.

Appendices

Credits

Josh Holland - Art Lead, 2D artwork
Ben Funderberg - 3D modeling
Tom Funderberg - 3D modeling
Hector Zhu - Splash Screen, Game Modes
Mikkel Sandberg - 3D modeling
Mitch Epeneter - Voice Acting
Ryan Schmitt - Deferred Rendering
Sam Thorn - Sound Lead

References

Mike McShaffry. 2003. *Game Coding Complete*. Paraglyph Publishing.

Daniel Sanchez-Crespo Dalmau and Daniel Sanchez-Crespo. 2003. *Core Techniques and Algorithms in Game Programming*. New Riders Games.

Randima Fernando. 2004. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education.

NVIDIA Corporation. 2008. *PhysX Documentation*.