Pirates Versus Ninjas: An OpenGL-Based Videogame Project

Nathan Kupp, Mihalis Maniatakos
Department of Electrical Engineering, Yale University, 10 Hillhouse Avenue, New Haven, CT 06520, USA

Abstract

The game "Pirates vs. Ninjas", documented in this report, is based on the popular internet meme. The gameplay is structured in the first person shooter genre. The user is able to select the class which he or she would like to play as, either "Pirate" or "Ninja", and then complete a level by defeating computer-controlled players of the opposing class in battle. The game includes several challenging computer graphics elements, including collision detection between projectiles launched by the player and enemy characters, several very complex objects with many thousands of polygons, and animation sequences for enemy characters.

1. Introduction

The game begins by initializing an OpenGL context within the Windows API. Instead of directly loading all graphics elements from their respective image and model files, we load only a splash graphic and mask to display a "Loading" screen. After this screen has been displayed, the remaining graphics elements are loaded into memory and the game state machine is entered at `GAMESTATE_SPLASH_START`.

The game state machine is shown in Figure 1. The majority of these states are responsible for blitting textures onto the screen in orthographic projection, producing a game interface.

In many of these states, alpha blending is used to give the illusion that the interface is floating above the game world, as can be seen in Figure 2. A function `CStaticImages::drawStatic(mask, image)` was implemented to draw static textures.

If the user selects the option "Start" at the splash screen, he or she is then presented with a choice to be either a pirate or ninja (in `GAMESTATE_SELECT`). This selection determines the behavior of the game in the following ways:

1) The player's weapon is either a gun or a throwing star, depending on whether the player selected to play as a pirate or ninja, respectively.
2) The enemies are ninjas if the player selected to play as a pirate, or pirates if the player selected to play as a ninja. These enemies are initially placed at random locations—near the Japanese tower, if the enemies are ninjas, or near the pirate ship, if the enemies are pirates.

The loading of enemy models is delayed until the player has made this selection. After the user presses return to select a character type, memory is allocated for an array of enemies, and the correct type of enemy Milkshape models are loaded. The game then begins.

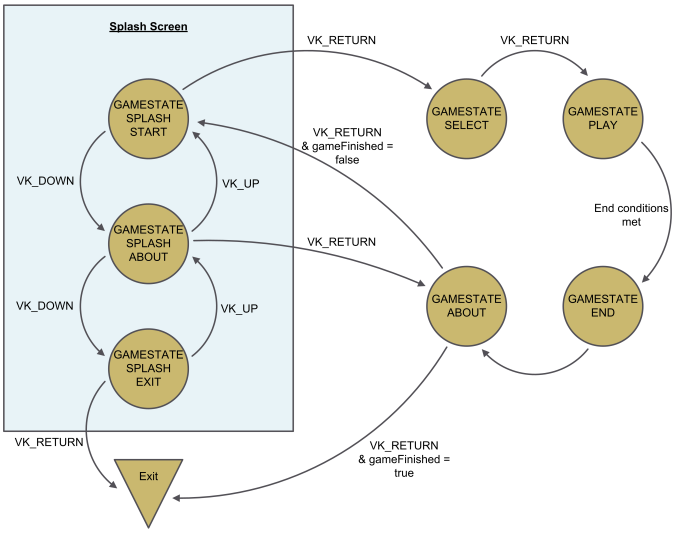


Fig. 1. Game State Machine

Fig. 2. Game Splash Screen

2. Gameplay

At this point, the player is put into the Pirates Vs. Ninjas world at a central location. There are two bases at different

parts of the map, densely populated with enemies. The objective of the game is to travel to the base of the opposing team and eliminate all the enemies encountered. This is achieved by firing the equipped weapon, which is either a gun or a throwing star, depending on the character class the player chose. When the player defeats all the enemies, he or she is presented with a "You Win!" screen, and the game terminates. If the players gets too close to the enemies and they are able to bring the player's health to zero, the player loses and is presented with a "You Lose." screen.

3. Class Structure

As can be seen from Figure 3, the programming structure is centered around a CGame class through which most of the game mechanics are governed.

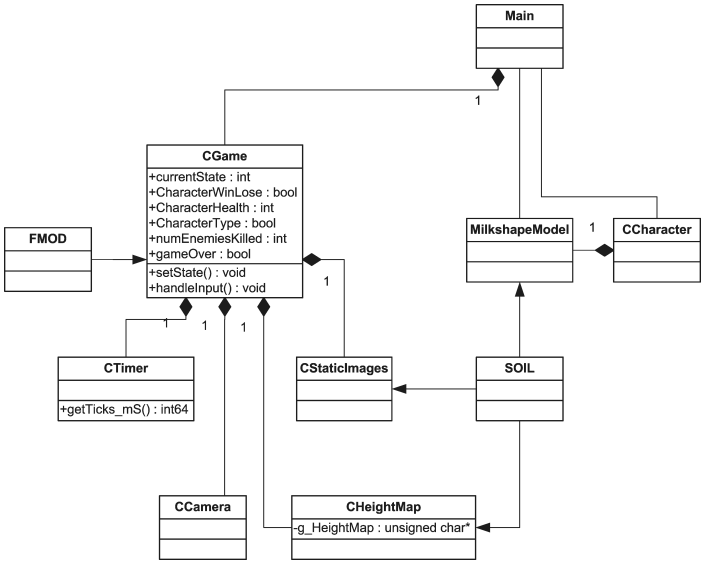


Fig. 3. Game Splash Screen

3.1. Game

Most importantly, the game state machine is managed in the CGame class with the variable `currentState` and the method `void setState(int newState)`. This method is called when keyboard input occurs, initiating state transitions based on the state machine shown in Figure 1. CGame subclasses FMOD, CTimer, CCamera, CHeightMap, and CStaticImages. Thus, by giving a class a pointer to CGame, the class can access much of the other game classes.

3.2. Sound

The FMOD library is used to implement game sounds. This library provides simple function calls to load and play back sounds of many formats, including the .mp3 and .wav formats. The library is used in the game in two ways—a streaming function is used to stream background music from sound files, and a sound clip function is used to play back sound clips based on in-game actions.

3.3. Time

The CTimer class was imported from a previous robotics project. We instance the class in CGame to provide a global time that can be accessed with `getTicks_ms()`. The CCharacter class uses the timer heavily to time actions.

3.4. Camera

The CCamera class implements a camera, adapted from code on the APRON tutorials site at `http://www.morrowland.com/apron/tut_gl.php`. The class originally provided camera movement on the xz-plane using the traditional WSAD-keys for forward/backwards/strafe and mouse input for rotation. This was modified to provide collision detection with the terrain and the player. As the functions for movement are contained in the CGame class, the CGame class passes a pointer to the heightmap to CCamera. Thus, whenever the camera is moved, the y-axis location of the camera is placed at a height slightly above the heightmap, giving the impression of the player walking around a world with complex terrain.

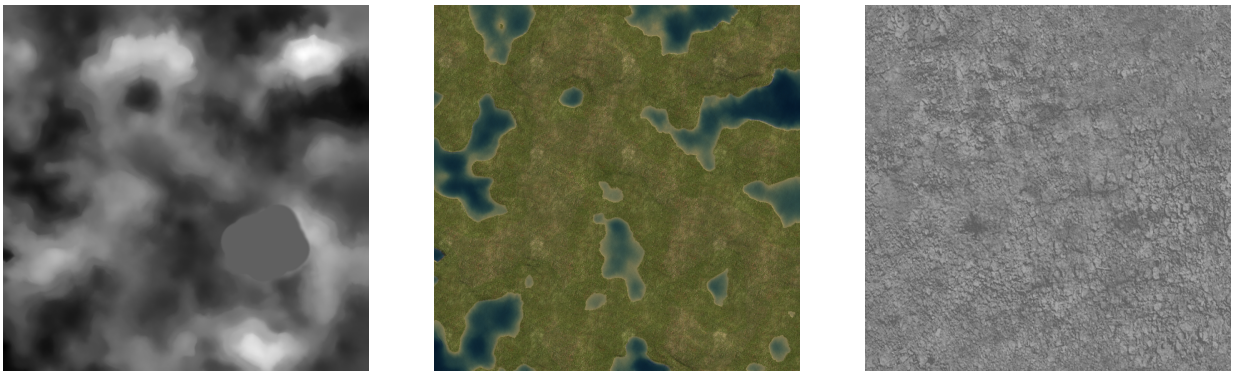
3.5. Heightmap

The heightmap class was written to import three graphics files and turn them into a rendered heightmap. This was achieved as follows. First, a heightmap and associated texture was generated using L3DT, a free terrain-generation program. This software provides several choices for the type of terrain to be created, and then generates a random terrain heightmap and texture, as can be seen in Figure 4(a) and 4(b). Note that the heightmap was hand-edited to provide a large flat area on which the ninja tower could be positioned in the game.

With the heightmap images generated, CHeightmap first loads the file `heightmap.png`. To render the heightmap, a grid resolution is defined to specify `GL_QUADS` in the xz-plane. For every vertex in this mesh, the `heightmap.png` file is parsed at a corresponding location and the grayscale color value is used to determine the appropriate y-axis location of the vertex—white being the highest possible point, and black the lowest.

Based on the defined grid resolution, the heightmap texture `terrain.png` is mapped to the entire grid mesh, by adapting the texture coordinates based on which quad is currently being drawn. Clearly, using this texture alone would result in a world terrain with a blurry appearance, as the texture resolution (1024x1024) is not high enough to ensure every quad is mapped with a high-resolution texture.

Thus, the OpenGL multi-texturing extensions were used to provide higher apparent resolution, by doubly-texturing every quad. While the original OpenGL specification only permitted



(a) Heightmap (b) Terrain (c) Terrain Detail

Fig. 4. Heightmap Images

a single texture to be mapped to a polygon, the OpenGL extensions permit up to 32 texturing units. Thus, when texturing the terrain we proceed as follows:

1) Activate `GL_TEXTURE0_ARB`.
2) Enable 2D texturing and bind `heightmap.png` to the texturing unit. Use a small piece of this texture.
3) Activate `GL_TEXTURE1_ARB`.
4) Enable 2D texturing and bind `detail.png` to the texturing unit. Fill the quad with this texture.
5) Draw the heightmap quad vertices using `glVertex` calls.

The effect of this technique is terrain that appears to be very detailed, without the overhead of loading a single extremely detailed texture, or a series of smaller textures that require sophisticated blending, as can be seen in Figure 5.

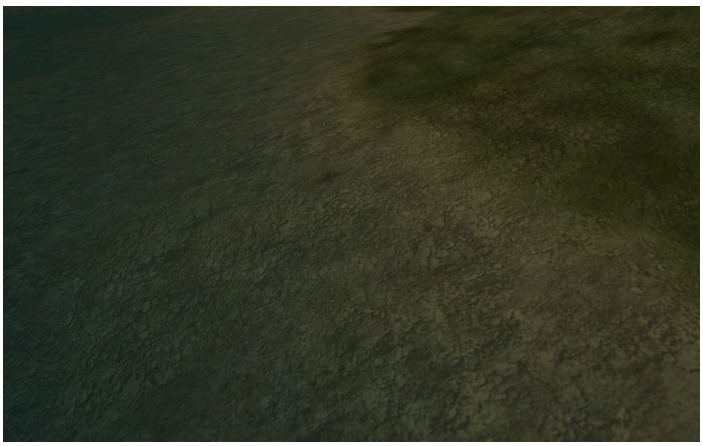


Fig. 5. Detailed Terrain

3.6. Texturing

The CStaticImages class is instanced for two purposes. First, all textures are loaded through this class when the game is initialized. The class uses the `SOIL_load_OGL_texture` method of the SOIL library to directly load the various images and store them as OpenGL textures.

Secondly, the class has several methods for drawing textures across the entire screen, using orthographic projection and alpha blending. Primarily, the method `drawStatic(mask, image)` is responsible for most of the 2D texture overlays, and is called by `drawGLStatics()` to overlay various textures based on the current state of the game. A function `drawCross()` draws a crosshair for targeting, and finally a function `drawHealthStatic()` draws a health indicator on the screen using a pair of images by blending them together. This graphic is drawn by accessing `CGame->CharacterHealth`, scaling it to the range $0.0 - 1.0$, and using this value as the alpha blend value when superimposing the two health images. The effect of this process is shown in Figure 6.

Fig. 6. Health Blending

3.7. Modeling and Animation

Models and animated characters are directly loaded from `*.ms3d` files using the MilkshapeModel class, written by Brett Porter. This class inherits from the base Model class, which handles the low-level reading of files and loading the triangle meshes. Milkshape models do not store textures internally, so the Model class was edited to use SOIL for loading the textures of Milkshape models. The class also handles animation, providing an `advanceAnimation()` method which performs skeletal animation using stored keyframes. As many animated

Milkshape models are freely available on the internet, this suited the needs of our game perfectly.

3.8. Collision Detection

In the game two different forms of collision detection are implemented: Collision of the player/models with the terrain and collision detection of the shooting object (star or bullet) with the enemy models. Collision detection with the terrain is straightforward, given the heightmap used for terrain generation. Each model's x and z coordinates (including the player) are given to the heightmap function, which will return the y coordinate at that point (since we defined the $(0, 1, 0)$ as the up vector).

To implement collision detection between enemy models and the projectile, a vertex by vertex overlapping test is performed. The collision test must be applied after the required transformations, such as scaling, rotation and translation, are applied. Checking every vertex of every model at every time is extremely inefficient, so bounding spheres were added to each model. Bounding spheres collision testing is very fast. So the vertex by vertex tests are performed only if the bounding spheres intersect.

Still, there is a problem with collision detection. The projectile's position is updated at each time unit. Since the projectile is moving fast, its position is moved at great distances at each time unit. Thus, a collision may be lost because of this—in fact, most collisions are missed. Thus, a prediction scheme was introduced to surpass this limitation, checking the projectile's position in much smaller steps. The shooting object is still drawn at the same intervals, but there are many positions between those points where collision tests are performed.

3.9. Non-Player Character AI

A state machine was implemented to provide simple enemy AI. For these characters, three states were created: `WANDER`,`DEFENSIVE`, and `AGGRESSIVE`. For the `WANDER` state, the enemy characters select an endpoint randomly in a square around their original position. They then create a vector $\mathbf{v} = \textbf{endPoint} - \textbf{startPoint}$, and use the global clock to move along this path using the equation $\textbf{currentPosition} = \textbf{startPosition} + \mathbf{v} \cdot \Delta t$. Whenever the enemy character arrives at a point within a defined radius, a new destination point is selected, and the process is repeated.

For the `DEFENSIVE` state, the enemy character does not move. However, when the human player moves within a certain radius of the enemy character, the attack animation is activated and `CGame->CharacterHealth` is decreased at the rate of 10 hitpoints/second, from an initial value of 100. If the player leaves the attack radius, the enemy character will return to an idle animation and remain stationary.

Finally, the `AGGRESSIVE` state is simply a superposition of the first two. By default, enemy characters wander around randomly. When the player is "spotted" by the enemy character, he will switch to the `DEFENSIVE` state and attack the player. If the player moves beyond the attack radius, the enemy character will resume wandering.