6.111 Final Project

Don Goldin
Mark Sullivan

Vertex

Overview

Asteroids is name many people associate with the dawn of gaming. The player has the ability to spin, shoot, and move forward. The player's goal is to avoid colliding with the asteroids on the playing field. Shooting the asteroids resulted in the substitution of several smaller asteroids in place of the big one. Video games have evolved a lot since then, but the ideas of *Asteroids* have been all but abandoned.

As with most classic games, *Asteroids* remakes are in no short supply. However, one comparatively recent game has successfully expanded on this formula. This game is called *Geometry Wars*, appropriately subtitled *Retro Evolved*. Similar to *Asteroids*, the player controls a ship, is able to shoot at any angle, and must avoid being hit by on-screen obstacles. Also, a 2D wireframe art style mimics the true vector graphics of the original. There are several differences, however, which distinguish *Geometry Wars.* The player is able to move and shoot independently. The player's input controls the player's velocity, instead of in the original where the input controlled the force and the ship kept its momentum. All enemies have different AI, whereas in the original most enemies were floating asteroids. This fresh take on an old formula was greeted by respectably high sales and reviews. Many of our ideas mirror the implementation of *Geometry Wars*, so that is the best existing game to use to get an idea for what this project aims to become.

The plan for this project is to implement a game similar to *Geometry Wars*. This will require the implementation of several pieces of functionality. A graphical component must be created. The plan is to preserve the aesthetic experience of *Geometry Wars*, so all in-game entities will be represented as transformable 2D wireframes. The player's avatar will be able to move and shoot independently. Because of this, the player needs an interface to perform each of these actions. For movement, a discrete interface should be permissible. A device such as a standard keyboard could be used for this task, via the arrow keys or the wasd keys. In order to give the player the ability to hit anything on screen, a less discrete means of interaction is desirable. A mouse is what is required at the simplest level. If time constraints permit, however, a theremin-like interface or visual input could be used to accomplish this task as well. The game logic will need to control the behavior of the three main classes of objects: the avatar, the shots, and the enemies. Different types of enemies will have different behavior and a different associated graphic representation. Enemies will be destroyed upon bullet collision, and the avatar will be destroyed upon collision with the enemy.

Modules

Input Module

This module is responsible for, at the simplest level, interpreting keyboard and mouse input and generating usable signals from them. In the event that it is possible to create a more complicated interface, this module will become much larger. This module should output signals corresponding to up, down, left, or right presses. Also, it should sensibly convert mouse input into an angle where the ship will aim its turret. This portion will likely require tweaking to get a good feel for it.

Game Module

This module is the central hub for the game logic. This will, as a whole, clock at 60Hz, triggered on the falling edge of the vsync signal. However, some parts will be pipelined, so it's not unforeseeable that this module will make use of the 65MHz clock as well. It takes input from the input module, and is able to decide what the avatar should do in response to that. Most other decisions are independent of user input.

It will contain an array of some size N, on the order of 64 as an estimate. Each element in the array will contain information about each entity in the game. An entity can be the player, a shot, or an enemy. The player, shots, and each different type of enemy will have a unique ID. Also, these entries will contain bits which correspond to the current position, position last frame, and angle of the entity. Therefore, every onscreen entity has a corresponding entry with a unique index (though not necessarily a unique ID). For simplicity of the collision module, it is possible that certain sections of the table will be reserved for particular classes. For example, perhaps we reserve index 0 for the avatar, indices 1:31 for the player's shots, and 32:63 for the enemies.

The AI handling and collision handling will be done by separate modules. Positions and rotations will be updated at the end of each frame, and collisions will be appropriately processed. The entity table will be sent over to the graphics module for visualization.

AI Module

Each non-avatar entry will be sent over from the game module one at a time. It may also require information about the avatar's current position, so that information will be given as well. This module then uses the ID of the entity to decide what its next position and rotation will be. It will then return that information to the game module. Should this require multiple clock cycles, this section could be pipelined, or, in the worst case, have an associated 'ready' signal when it is done processing a particular entry.

Random Number Generator

Enemy spawns and perhaps AI behavior would desirable have some associated randomness. For this purpose, our project will require a pseudo random number generator. If a

simple approach such as the middle-square method is adequate, that will be used. Otherwise, more advanced and accurate pseudo-random algorithms will be investigated. It is likely that a single random number module is sufficient, since the way the AI module and game module interpret the random number will be different.

Collision Module and Unit Sizes/Shapes

The purpose of the collision module is to determine which objects are colliding. Following the design, the types of relevant collisions are enemy-bullet or enemy-avatar. It shall be assumed that there can only be one meaningful collision for the avatar and for each of the bullets. Therefore, there can be an array with (1 + max_bullets) elements. If a collision is true for any of them, then the unique enemy index is returned. If there is no collision, there will be a zero for that spot in the array, since zero will never be an enemy index if we reserve it for the avatar. There will then be the actual collision detection part. It is very likely that this will need to be judiciously pipelined, since there must be (1 + max_bullets)*(max_enemies) collision comparisons. If all enemies are generally circular, then circular collision detection may be accurate enough. Otherwise, bounding box collisions or some other type of collision detection would have to be used. This module would need some way of acquiring collision boundary information based on the enemy type, hence the existence of a Unit Sizes/Shapes Module, which is simply a look-up table of the relevant information.

Screen Buffer Module

The screen buffer module consists of two 1024x768 buffers of pixels and some associated logic. Each frame, one buffer is displayed to the screen while the other is erased and presented to the wireframe module for drawing. It may not be possible to perform both the erasure and the drawing in one 60hz cycle; if so, either an in-RAM buffer stage or a reduced framerate will be necessary.

Wireframe Module

The wireframe module receives each frame a list of line segments to draw to the screen and their associated colors. It iterates through the segments, setting the offscreen pixels on each segment to the associated color. This may sound simple enough, but it will probably be the most complicated stage of the graphics pipeline; drawing a segment between any two points is much more difficult than drawing a sprite. Time and resources permitting, we might add anti-aliasing and/or alpha blending to this module.

Graphics Controller Module

At the beginning of each frame, the graphics controller reads from the game module the position, rotation, and visibility bit of each entity. It looks up the model of each visible entity (a list of line segments) and performs translation and rotation on it using fixed-point arithmetic. It then passes the transformed line segments to the wireframe module to be drawn.
This module will require the use of trigonometric functions in order to handle rotation; most likely these functions will be implemented as lookup tables with interpolation.

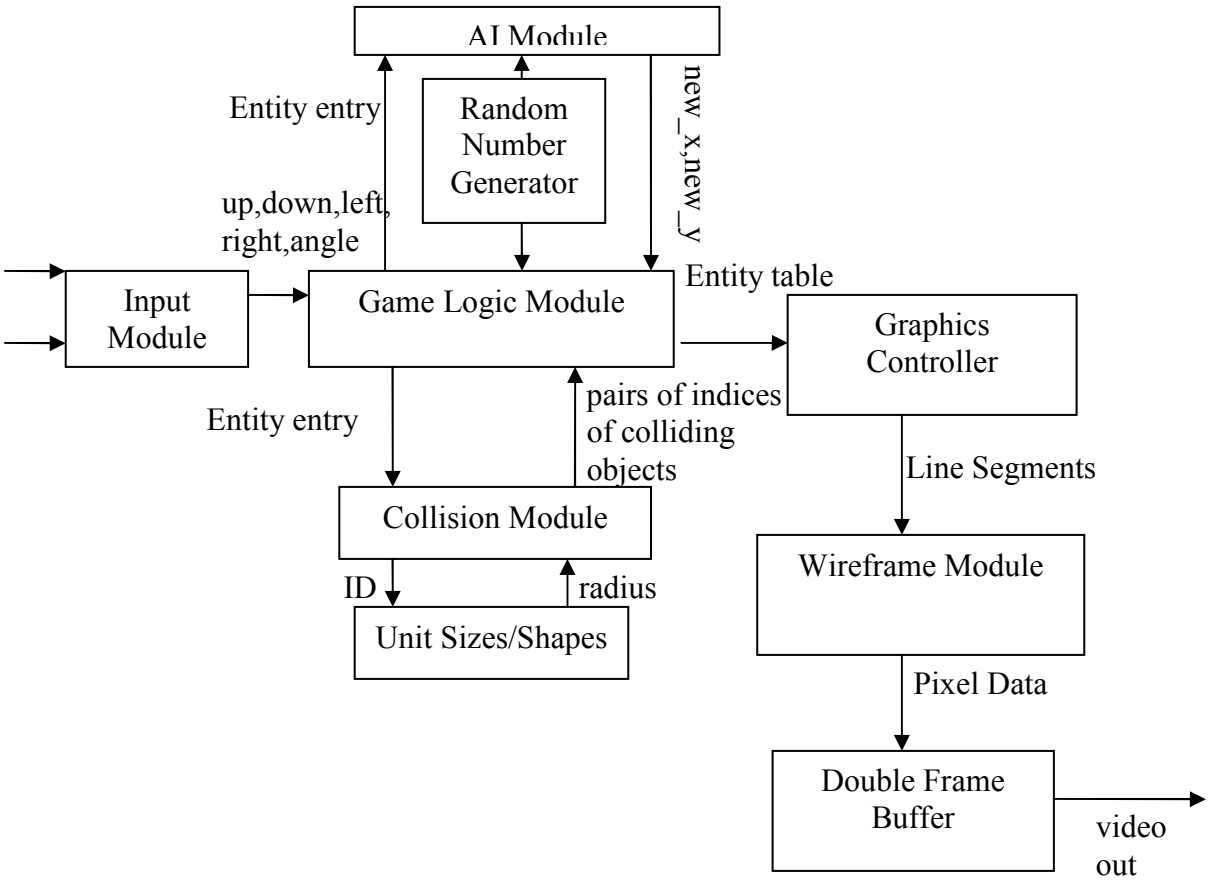

Figure 1: Block Diagram