Spatial Invaders

Game Design Document

Goal

Create a project for the Bacon Game Jam '15 48 hour game jam. Theme is “dimension.”

Overview

Use a 3D engine to create a game that requires a user to think in 3D with 2D visualization. Bonus: learn the Blender game engine well enough to produce a complete game. Game jam rules require all work to be completed within 48 hours. As a special challenge, I'll build the game in a hotel room in China with limited Internet access and no Google. (OK, I'm going to have to VPN a bit.)

Game Description:

Spatial Invaders is a twist on the standard “shoot aliens who are slowly falling from the sky” theme. The standard version of space invaders is strictly 2D with a side view and aliens moving across and down the screen. My version will involve a single alien moving erratically in two dimensions (X and Y) using a form of the random walk algorithm. The alien will periodically move down closer to the earth. If the alien reaches a certain minimal threshold, the aliens have reached the planet and the player loses.
The player will operate a small turret with keyboard (or perhaps joystick) controls. The player will maneuver underneath the alien and fire straight up. Bullets will be visible and will conform to gravity and collision. If a bullet hits the alien, it is knocked back up into the sky by a small amount. If the player is able to force the alien to a specific altitude, the player will win, because the alien has chosen to go home.

Game Assets:

Player: simple 3D sprite. Should look like a turret facing upwards. No animation needed.
Bullet: build offstage for dynamic creation. Very simple sprite because many may be on screen at one time. Bullet should die when it hits the ground, to prevent over-saturation of memory.
Alien: Standard UFO design. Simple wobbly animation should be enough.
Environment: Since I'll be using fixed cameras, a very simple environment should be sufficient – just two UV-mapped planes.
Audio: We'll need simple digital sounds to indicate the alien moving down (negative sound) and the alien being hit (positive sound) BFXR should be sufficient.
UV maps: Very simple UV maps will be used to add style. A basic ground and sky map should be more than enough to add feeling. Likely these will be shadeless to add more light to the game.
Lighting: A simple sun with basic point fills should be sufficient.

UI:

A key feature of this game will be the split-screen UI. The user will see two screens at once (something like the Nintendo DS) The top screen will be a top-down satellite view of the playing area. This will show the X and Y positions of both the player and the alien, but it will not have an indication of the alien's altitude. The bottom screen will be a more traditional space invaders side view. This view will show X and Z positions of the alien and the player, but no Y information. The player will need to integrate the two views to get a sense of the game situation.
An altimeter will indicate the current altitude of the alien (probably a standard text field.) This will be the primary progress indicator, apart from the visual cues from the split screen.
Viewports.py from tutorialsforblender.com looks like a good starting point for the two screen mechanism.

Game Object Mechanics

Player Mechanics:
Player controlled by WASD keys to move on ground. Player can be static or rigid body, no need for more complex physics. Player will not move on Z axis, and will not respond to any collisions, so no special collision model will be necessary. Space bar will fire the bullet, but the space bar is read by the player object because there will be multiple bullets. Player is persistent, will never be created or destroyed during the game. Most player motion will be performed in python script to limit motion to proscribed playing area.

Bullet Mechanics:
Bullet is a simple icosphere created in a different layer. Bullet will need dynamic physics model. Default mass should be fine as bullet will be launched with a large linear velocity in positive Z.
Bullet will disappear upon contact with alien or ground. Begin with standard collision sphere and adjust as necessary. Collision sphere will be larger than bullet radius, but this may be adjusted for difficulty level. Bullet will need custom material (bulletMat) for collision detection. Logic bricks will likely be enough for bullet management.

Alien Mechanics:
Alien will use a random walk algorithm to move around the XY plane. Random DX and DY within a small range (initially -.2 to .2) will be added to X and Y in a Python script. Use a small random value to check whether the alien moves downward on any given frame. (Increase this likelihood to make the alien more aggressive and the game more difficult.) If alien detects collision with bullet, the alien will move upwards along Z axis a small amount. Any alien vertical movement will be accompanied by a reinforcing sound effect and a change in the altimeter readout. If alien moves below a certain threshold, the game lose condition is triggered. If the alien moves above another threshold, game win will be triggered. Message-passing will trigger state changes (See state transition section.)

States and Transitions

Launching the game goes immediately into gameplay state, with all inputs available. An

instruction screen may be added if time allows, but initial instructions will be in a simple text file attached to blender file. Winning the game triggers a win state, and losing the game triggers a lose state. Each of these states will involve freezing the gameplay screen and placing a new scene overlay indicating the win or loss. The overlay states will also allow user to restart the game or quit using keyboard commands. Quit will immediately quit the game, and restart will send a restart message to all sprites allowing them to return to a gameplay condition.
Restarting the enemy simply causes it to move to the initial altitude. Restarting the player will place the player at its original starting place and turn the main scene back on, returning the game to the play state.
Message-passing will be heavily utilized in the state transition process.

Milestone plan

Viewports – The novel multi-view system is completely new to me and key to the game. If I can't get this working quickly, I'll need to rethink the game completely, so this needs to be the first step.

Basic models and scenery – Very simplistic player and enemy model, ground and background. No UV maps or in-depth animation yet. This will be added later as time allows.

Player motion – Use a python script to get player motion with boundaries. Script makes flexible boundaries possible and also allows for speed to be changed easily as a difficulty mechanic.

Enemy motion random walk – The first pass on enemy motion will be the basic random walk. Use a python script to get appropriate random behavior (adding between -.2 and .2 to dx and dy each frame.) May need to adjust for maximum speed. Bounce off of edges? Edge behavior could be a tricky dynamic as the AI will be easy to shoot down if it gets trapped in a corner. Bug or feature? Save up and down behavior for a later step.

Bullet dynamics – Create dynamically created bullets flying from player with high positive delta-z. Add collision with enemy (delete bullet and add sound effect) and collision with ground (delete bullet.) Logic bricks may be sufficient, but use a script if necessary.

Enemy altitude – Enemy – bullet collision will cause the enemy to move up in z. Use a script for this? Also add a fixed text box to show the current altitude and modify the text box to indicate the altitude. Add the random behavior to drop altitude at rare intervals into the move enemy script.

State management – Create states for beginning and end of game behavior, and implement them with message-passing. Python scripts may be necessary but they should not be complex.

Tuning

Tune for gameplay. Adjust randomness of enemy motion, speed of enemy and player, speed of bullets, bullet collision radius, how often the enemy moves down. Add improved UV maps, character models, and animations as time allows. Consider powerups, limited ammo, and bombs. Verify in-code documentation, and add gameplay documentation. Create executable for submission to contest.