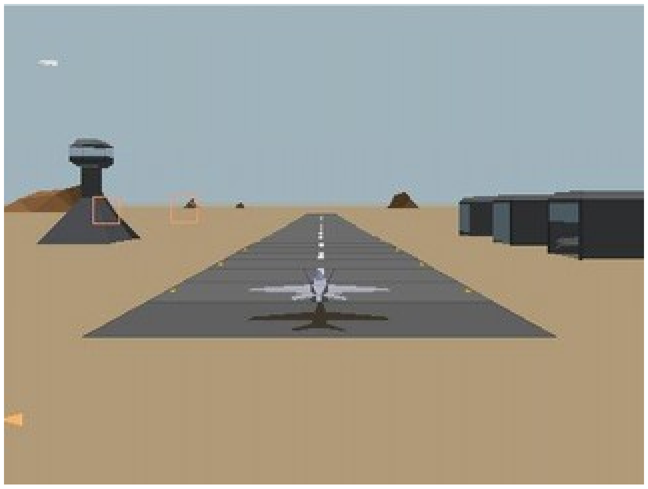

Design Document for:

Project: SE-800

Space Combat Evolved

Version # 3.00
Tuesday, April 22, 2014

Table of Contents

Design History

This is a brief explanation of the history of this document.

The initial version of the design document serves as a template for further specification and clarification.

Game Overview

Common Questions

What is the game?

SE-800 is a multiplayer spaceship-shooter game in which players control a squadron of spaceships, shooting and evading asteroids and enemy vessels.

Why create this game?

We believe this game will provide graphical, AI, and physics engine-related challenges which will provide a valuable learning opportunity, and yet still are feasible within the scope and time restraints of the Game Construction class.

Where does the game take place?

The game takes place in a fictional environment resembling outer space. In this no-gravity environment, player spaceships are affronted with the constant threat of asteroid collisions and attacks from enemy ships. We may potentially choose to insert planets into the environment as a source of gravitational pull.

What do I control?

Players will be assigned a number of ships per their team, and they will have direct control over one ship at a time. Each ship which they are not immediately controlling will be directed by AI. Players will also have the ability to assume direct control of any ship on their team.

What is the main focus?

The following gameplay modes will be implemented which correspond to different win-case scenarios:

- Last man standing: Players shoot and evade asteroids and enemy ships with the main focus being to destroy all enemy ships. The player with the last ships remaining is the winner.
- Time dependent kill count: Gameplay has a finite time and ships are able to regenerate after destruction. The player who is able to destroy the greatest amount of enemy ships before the time is up is the winner.
- Time dependent item collection: Gameplay has a finite time and items are available for collection throughout the environment. The player who is able to collect the most items before the time is up is the winner.
- Dogfight: PvP match, one ship only.
- Single-Player: Player does either of these game modes against a fully AI controlled opponent.

Feature Set

General Assets

3D Models/Textures:

- Ships: Team A/B Fighter Ship Design, Team A/B Carrier Ship Design
- Environment: Asteroid Model, Planet Model, Power-up Model
- Ambient/Background Design (i.e. Space, Distant Solar systems)
- Projectiles: Bullets, Missiles

Multiplayer Features

Two-player with NPC Teammates
Network play between two computers
Multiple Competitive Modes

Gameplay

Maneuverability in a 3D space:

- Forward Acceleration/Braking Control
- Roll Axis Rotation
- Pitch Axis Rotation
- Weapon Firing and Switching

Objective Based Gameplay:

- 1v1
- Highest Kill Count
- Last Ship(s) Standing

The Game World: Design Features

Overview

Players compete in a fixed volume outer space environment. Upon reaching the boundaries of the map, objects and player perspective transition to the opposite end of the map, leading to a continuous game world.

World Grid Mapping

In order to best optimize processing performance and gameplay speed, will perform dependency calculations (i.e. physics, AI) on neighboring and current region basis: will calculate based on nearest and present grid space.

Environmental Obstacle Motion

Asteroids will rotate through space and flow forward with constant direction velocity unless destroyed. Upon being destroyed, asteroid objects are flagged and removed from the game state.

Power-Ups

There are a variety of objects that are non-damaging collisions such that they change the characteristics of ship objects. Such changes may include speed-boost, health regeneration, missiles, and/or point objects (contribute to overall score).

Time

The evolution of the game space over time will be handled by the physics engine. The physics engine will move non-ship objects in the game space such as asteroids over time and a log will be kept of object manipulation through a game match. Time may potentially play a factor in terminating the game state and also determining victory.

Rendering System

Overview

Graphics will be generated at 60fps through use of the OpenSceneGraph graphics engine and programming for this library will be done in C++. Model creation will be accomplished through the use of Blender and other not yet determined software.

2D/3D Rendering

Background will be a 2D rendered texture mapped onto the boundaries of the gamespace. Models will be 3D rendered and mapped to design and bump textures, and the final determination of pixel colors including highlight and shadows will be calculated with a Phong shader. We will also be using shadow mapping, taking an orthographic projection from the directional light’s origin direction to determine which pixels are being shaded.

Camera

Overview

Camera will follow behind the player ship along a fixed axis and will adjust to each type of ship motion differently.

Roll Camera Shifting

Camera will rotate in constant time with any changes in roll rotation of the ship object.

Pitch Camera Shifting

Camera will rotate in constant time with any changes in pitch rotation of the ship object. Camera tolerance may be added in the future to limit disorientation

Game Engine

Overview

We're likely going to use the g3d engine the enable easier rendering of models that we'll create in Blender. This will give us more time to explore how we will generating and displaying the world in depth. Adding on to that, we'll be creating a world representation for the AI and the physics engine to interact with.

Game Engine Detail #1

The game engine will keep track of everything in the world like asteroid and player locations.

Collision Detection

We will implement a rudimentary form of collision detection in which objects can't enter within a sphere that is just large enough to contain other objects. These spheres will allow us to calculate minimum distance between two objects, and when the bounding spheres overlap, the objects collide.

Lighting Models

Overview

We will be using Phong lighting to illuminate each object in the world. We will use a fixed directional lighting the illuminate all objects in the world. In the future we may be able to add point lights such as suns actually within the environment, but that is not currently within the scope of the project.

The World Layout

Overview

World will be an open 3D space filled with spherical asteroid models and powerups dispersed randomly to fill the space. Ships will navigate this space with obstacles and will be treated as smaller spherical ship models for collision detection. Space will consist of a boxlike volume.

User Interface

Overview

Players will control the game state through the use of a controller or mouse and keyboard.

HUD Display

HUD will be overlaid on player perspective and transparent. This will include a radar that detects objects in the vicinity and indicates object type, a player ship health bar that will indicate how much damage has been taken, current weapon stock, and target object health.

Menu Select

There will be a menu upon starting up the program. It will allow for single-player or multiplayer and then upon accessing either menu, will allow for selection of game-type.

Controller Support

Users will be able to use a mouse and keyboard to control movement and/or potentially an Xbox 360 controller.

Weapons

Overview

All ships will carry an unlimited amount of bullets from the start of ship life and a limited stock missiles. Bombs will also be a weapon type, although will have to be picked up. Different objects will have different health amounts, allowing some objects to take more damage.

Bullets

Bullets will appear as simple laser objects and will deal the minimum amount of damage out of destructible objects. However, they will have a faster fire rate than both bombs and missiles.

Missiles

Missiles will appear as rendered objects and will be bigger than the simple bullet object. They will deal more damage and have a significantly slower fire rate.

Bombs

Damage will be dependent on how close object is to center of source. They will have a timed-delay upon release for detonation. If object collides with bomb before detonation, bomb will detonate. Bombs will have the capacity to deal the most amount of damage.

Multiplayer Support

Overview

Describe how the multiplayer game will work in a few sentences and then go into details below.

Max Players

Only two players can play at once.

Servers

The game uses a client-server model with host migration.

Customization

Customization of ships through different textures is planned as added functionality.

AI

Overview

Since we envision this game to involve squadrons of fighter-ships engaged in large dogfights, the behavior of computer-controlled ships is important for gameplay. AI ships ought to have squadron-like behavior: following a leader ship (e.g. player), protecting the leader, attacking enemies in coordination, etc. As such, we will use a leader/follower group paradigm.

Leaders, Followers, and Groups

Each ship will belong to a group (have a group_id). Groups will have at least one Leader ship (possibly the player) and several Follower ships. The Follower's actions are heavily determined by the Leader. The Leader "thinks" for the Follower, doing most of the strategic AI computations; the Leader might determine attack targets, group size, pathfinding, attack timing, etc. The Follower will execute simpler actions such as follow the leader, attack nearby enemies, stay near allies, etc. Follower behavior will depend heavily on the group's status.

Alternative Paradigm: Tree of Command
An alternative paradigm would be a tree of command, where squadrons of ships would be organized like trees, with the Leader at the root node, and Followers as leaves. This way, the Group is the entire tree, and the Follower's behavior is dictated/inherited from its mother node.

Behavior

AI behavior will be recombinations of the "pure" behaviors described below. AI might have a priority list, or some linear combination of priorities, etc. The behaviors described below offer an schematic.

- Seek(location)
  Move towards target location
    - *Note: Seek(location, velocity)? the AI should be concerned about the velocity it arrives at the target point with*
- Random Walk
  Move erratically
    - *Note: Seek(random location)*
- Patrol(area)
    - Random Walk and Free Fire within a region

- Free Fire
  Attack nearby enemy/neutral targets.
    - *Note: maybe Focus Fire(all nearby targets). Leash Range?*
- Focus Fire(target)
  Attack the target (e.g. ship or asteroid)

- Flee(location/target)
  Fly to avoid the location/target. Non-combative

- Strafe(location/target)
  Move erratically tangential to the target to avoid bullets

Leader/Follower Behavior
Some of the actions above will be restricted to Leaders Only, others might be used by both. Follower behavior will be heavily influenced by the Group's status and the Leader's behavior.

The Leader might directly think for the follower; that is, the Leader might dictate what behaviors the Followers use.

Offense/Defense Behavior

The game mode has not yet been set in stone, but will probably involve two opposed bases which continually spawn ships. As such, we might want the AI to have both offensive and defensive behavior. That is, some AI will naturally defend the home base, and some AI will naturally attempt to attack the opponent. ("naturally" = without the player's direction)

AI Methods

We have to further flesh out how the AI will conceive of movement. Since we have a whole physics engine going, the physics state (position, orientation, velocity) is truly the ship's state in the space; position alone is not sufficient. The methods for pathfinding or obstacle avoidance will have to take into account the physics engine. Perhaps the AI will have a crude approximation of the physics engine.

AI firing/targeting, what weapon type to use, etc. is also an issue.

Input/Output (Data ⇒ AI ⇒ Control)

AI computations will be made ~2 Hz. AI difficulty may be set by clock speed (e.g. smart AI recalculates at 5 Hz, dumb AI at 1 Hz)

Data ⇒ AI

The AI has a "vision radius" around it which it is knowledgeable of other entities (ships, asteroids, etc.). Objects outside the vision radius do not factor into the AI's calculations. The AI vision range probably ought to be equal to the player vision range. Within the vision range, the AI will be able to see objects' position, velocity, team_id, etc,

AI ⇒ Control

AI will write to Control using the same types of actions that the Player Controller has; accelerate, turn, fire1, fire2, etc.

Game Architecture Specifics

World State

The World State will be contained in the Physics/Game Engine. That is, all of the objects which constitute the World State will be a part of the Physics/Game Engine class.
The World State contains the following objects:

- List of Grid objects

    - Where a Grid Object contains a hash of Asteroid and Ship objects which are contained within it
- Queue of graphical events which have not yet been rendered (i.e. ship destruction, ship shooting)

The World State can be queried with the following public functions:

- getNeighbors
    - Given an object ID of a user-controlled ship to be rendered, returns a list of surrounding objects which will also be rendered.
- getAllObj
    - Returns a list of all objects (including ships, asteroids, and bullets)

Physics/Game Engine

The Physics/Game Engine will be completed by Minke Zhang and Steven Garcia.

The Physics/Game Engine interfaces with the rest of the game components as follows:

- World State
    - Updates World State after every cycle
- Graphics Engine
    - Sends list of objects to render
    - Sends list of graphical events to render (i.e. ship destruction)
- AI Engine
    - Receives queue of AI actions (movement and attack triggers of all AI ships)
- UI
    - Receives queue of UI input (keyboard/mouse/controller movement)
- Network
    - Physics/Game Engine is started by Server Engine, which is in turn ignited by the Network spawning the Server Engine

Upon receiving movement triggers from AI and user players, the Physics/Game engine calculates any changes in acceleration, velocity, rotation, direction, and position for each object. Objects are then updated in the World State. If user player triggered movement, user player's Graphic Engine is sent a list of updated objects to render.

Upon receiving shooting triggers from AI and user players, the Physics/Game engine adds a graphical shooting event to the graphical events queue. If user player triggered shooting, user player's Graphic Engine is sent a queue of updated graphical events to render. Physics/Game engine then calculates whether or not the ship's projectile intersects with the bounding sphere of another ship or asteroid. If an intersection occurs, the engine determines whether or not the strength of the projectile is enough to destroy the object. If it is, the Physics/Game Engine adds an object destruction event to the graphical events queue and passes this on to the user player's Graphic Engine.

Events cannot be removed from the events queue until the Physics/Game engine has received an acknowledgement from the clients that these events have been rendered.

Graphics Engine

The Graphics Engine will be completed by Justin Lang and Laura Macaddino.

The Graphics Engine consists of three main functions with the following features:

- Ignite
  - Receives player ship data from Physics/Game Engine
  - Draws the world
    - World is rendered using OpenSceneGraph
  - Sets fixed point lighting
    - Lighting is set using OpenSceneGraph
- Cycle
  - Receives a list of asteroids and ships to render as well as a list of graphical events (i.e. ship destruction) from Physics/Game Engine
  - Saves this list of new objects to be rendered
  - Updates the camera based on the new position of the player ship
    - If this is not the first cycle, camera position is interpolated between old camera coordinates and new camera coordinates
    - Camera is updated using OpenSceneGraph
  - Draws the objects to be rendered based on the list of new objects to be rendered
    - First checks the graphical events queue for unrendered events (comparing event IDs with a list containing IDs of already-rendered events)
      - Renders any unrendered graphical events and stores event IDs in list of already-rendered events
      - Sends event rendered acknowledgement message to Physics/Game Engine
    - If this is not the first cycle a specific object has been rendered, object position is interpolated between old coordinates and new coordinates
    - Objects are rendered using OpenSceneGraph
  - Shading is completed for each object using OpenSceneGraph
  - A list of all objects rendered during this cycle are stored in a list of objects which have just been rendered during the last cycle. This is done for camera/object position interpolation purposes.
- Shutdown

The Graphics Engine interfaces with the Physics/Game Engine in the following manner:

a) During game start, the Physics/Game Engine assigns a client with a player ship object. This object is stored both in the server's world state and in the client Graphics Engine.

b) On a 60 Hz timeline, the Physics/Game Engine searches for the coordinates of each player ship and calculates the map grids each player should render based on the position of their player ships found in the world state. It then pulls from the World State a list of the ship and asteroid objects contained within these grids, and it passes a list of ships to be rendered and a list of asteroids to be rendered to each respective client. It also sends over a queue of graphical events to be rendered (such as object explosion, shooting, etc).

c) If a UI input event (keyboard/mouse/controller movement) has been triggered, the Graphics Engine will send the UI Engine its player ship ID. The UI Engine then sends the Physics/Game Engine an update request. This request will consist of ID of the player ship and the UI input data. The Physics/Game Engine then forward the Graphics Engine a list of ships and asteroids to be rendered as well as a graphical events queue based on the same strategy described in part b).

d) Upon rendering a graphical event from the graphical events queue, the client sends the Graphic/Physics engine an ACK message consisting of the graphical event ID. It then stores the event ID in a list containing the IDs of of already-rendered events.

AI Engine

The AI Engine will be completed by Eric Guan and Justin Lang.

AI functionality is described in the previous section. The AI interfaces as follows:

Data ⇒ AI

- Receives data about the world state (nearby objects, location of home base / enemy base, etc.)
  - get_objectives()
    - returns list of big game objects (e.g. enemy base location, player location)
  - get_ally_objects()
    - returns list of all allied ships or objects. For flocking behavior, strategy
  - get_nearby_objects(vec position)
    - returns list of nearby asteroids, enemy ships, ally ships. For precise movement/tactics, calculating what angle to turn, fire, etc.

AI ⇒ Control

- Sends actions to Physics/Game engine by sending it movement and shooting behavior
  - AI has access to the same basic actions that the player does.
    - e.g. thrust, turn, fire1, fire2, etc.

  - Like the UI control, actions are placed into a buffer which is checked/processed/cleared at 60 Hz.

AIs will either be Leaders or Followers to reduce computing load (see above section for details). The general structure of both AIs will be similar, but Followers will have significantly fewer data calls and computations.
AI behavior will be dictated by a finite state machine. On the broadest level, the AI will be in a Strategy state which orients it towards a particular objective (e.g. Attack, Defend). AI will switch between Strategy states infrequently. Given a Strategy, there is also a FSM which sets the Tactics state (e.g. attack nearby enemies, patrol area, flee towards nearby Leader) to determine what action the AI is attempting to execute. Given a Tactics state, the AI will perform computations to determine how to execute the Tactic (e.g. pathfinding, aim & fire, flee & strafe). This yields the Action that the AI will feed to Control.

Data ⇒ Strategy ⇒ Tactic ⇒ Action ⇒ Control

The variables that the AI will use to determine its state will include:
- position, velocity
  - pathfinding, how to reach a particular direction/orientation
- nearby allies/enemies
  - to determine aggressive/defensive behavior

Network
The Network will be completed by Seun Ogedengbe and Laura Macaddino.

Our Network architecture causes game components to be divided into two discrete categories:
- Client
  - Graphics Engine
  - UI
- Server
  - Physics/Game Engine/World State
  - AI Engine

The network allows the listed architectural components of the the client and the server to interact with each other over TCP and UDP.

When a user selects which type of game he or she would like to begin, the network decides which machine will contain the server.
- If the player begins single-player mode, the network handles the spawning of a server on the user’s machine.
- If the player begins multiplayer mode, the network assumes the server is already present on a dedicated machine and attempts to connect to this machine.

The server engine contains the following tasks:

- Listen on socket for client connection
- Spawns a thread for each client connection
    - Each worker thread contains a copy of a global Physics/Game Engine/World State
    - Each worker thread listens for client UI input and updates World State accordingly (using locks)
- Upon one client connection for singleplayer mode or two client connections for multiplayer mode, ignites the scheduler for the AI and Physics/Game Engine. Sends player ship data to respective clients.

Following a game connect, the network provides a series of utility functions which allow for communication between the client and server components.
The client uses the network to send data to the server as follows:

- UI
    - Sends UI input (keyboard/mouse/controller input) alongside player ship ID to Physics/Game Engine

The server uses the network to send data to the client as follows:

- Physics/Game Engine
    - Upon client connect, sends client player ship object data to the Graphics Engine (this is used to identify a client)
    - Sends list of graphical events to be rendered by Graphics Engine (i.e. ship destruction, ship shooting, etc)
    - Sends list of asteroids and ship objects to render to the Graphics Engine

We currently plan on using TCP for each of the above-listed client-server interactions. If TCP proves to threatening gameplay with low latency, we may switch to UDP whenever the Physics/Game Engine sends a list of objects and graphical events for the Graphics Engine to render.

UI

The UI will be completed by Steven Garcia and Seun Ogedengbe.

- When a user attempts to open the game, OpenSceneGraph will be used to generate a player window.
- User will be directed to a splash screen where he/she can specify the game mode (i.e. Single player mode, one of the aforementioned multiplayer modes).
- Upon selecting player mode, UI forwards network instructions on how server should be fetched. Upon selecting single player mode, network spawns a server engine on the

user's machine. Upon selecting multiplayer mode, network connects to a presumably preexisting server.

- Upon game connect, Graphics Engine scheduler is ignited and UI is forwarded the graphical output of the Graphics Engine. Game-specific keyboard/mouse/controller listener is also activated.
- While game is connected, UI cycle consists of:
  - Render the graphical output of Graphics Engine
  - For each keyboard/mouse/controller input event, UI obtains the client player ship ID from the Graphics Engine, packages ship ID and input event together, and sends this data to the Physics/Game Engine. This serves as an update request for the Graphics Engine.

Audio

Audio functionality will be completed by Steven Garcia and Laura Macaddino.

We plan on using the openAL library to facilitate audio generation.

Audio output will consist of ambient background noise, which is constantly looping throughout gameplay, and sound effects which correspond to graphical events (such as object destruction and shooting).

The audio state will consist of multiple sources of audio, which correspond to objects which are rendered per graphics engine cycle.
For each graphics engine cycle, the following audio-related tasks are completed:

- Determines which objects will be rendered
- Determines whether or not an audio source already exists corresponding to each object. If source does not exist for an object, creates the a new audio source for the object.
- If any graphical events are rendered, pairs the object involved with the graphical event to its corresponding audio source. Updates the audio source with object's position and velocity data and plays sound effect which matches the graphical event.

Asset Creation

Asset Creation will be completed by Steven Garcia and Laura Macaddino.

We plan on using Blender for ship and asteroid model construction.
We plan on using Gimp for texture construction.

Audio files will most likely be pulled from an open source of audio clips, such as The Free Sound Project.