PART IV

**Comp 490 Software Design**

****

**California State University Northridge**

**Computer Science Department**

**Scorch3d Earth**

By: Matthew Hoggan

Hangul Lim

David Schaffer

Davit Stepanyan

Author: Matthew Hoggan

Creation Date: 9/25/2010

Last Revised: 5/11/2011

Version: 1772

Contents

[Introduction 3](#_Toc290338112)

[System Overview 3](#_Toc290338113)

[Design Considerations 3](#_Toc290338114)

[*Assumptions and Dependencies* 3](#_Toc290338115)

[General Constraints 4](#_Toc290338116)

[System Architecture 4](#_Toc290338117)

[*Subsystem Architecture* 5](#_Toc290338118)

[Detailed System/Subsystem Design 5](#_Toc290338119)

[*Menu System* 5](#_Toc290338120)

[*Play-Game System* 7](#_Toc290338121)

[Game logic: 7](#_Toc290338122)

[Terrain: 8](#_Toc290338123)

[Player: 10](#_Toc290338124)

[Projectile: 11](#_Toc290338125)

[Skybox: 11](#_Toc290338126)

# [Introduction](http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC4)

The purpose of this document is to give a more detailed description of the systems and overall architecture for the “Scorch3d Earth” software project. It will provide a system overview, explain design considerations, and provide details on individual components that will make up our systems.

It is recommended to read the SRS document prior to reading this document to get a basic understanding of the system architecture.

This document is a work-in-progress and may contain insufficient details which will be filled in as the system design matures.

# [System Overview](http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC5)

Unlike other software, video games require a special architectural approach. Players are usually placed into a game world where objects move, often without any input from the player. Traditional event-driven architecture simply cannot model a dynamic 3d world properly. The solution is the “game loop”. The game loop is basically a block of code that is repeated. Each time the loop is run, the game does any calculations to update the objects in the game, and then draws the scene for that frame. When this is done many times per second (games typically run at 60 frames per second) the illusion of dynamic motion is achieved. This game loop is the main system that ties the whole project together.

The project calls for two distinct systems to be implemented. First, there is a menu system which allows the user to configure their game as they see fit. Second, there is the play-game system which models the 3d world and gives control of the tanks to the players; this is the actual game part. These two systems will be accessed sequentially, never concurrently, so both systems can be added to the game loop and then separated by a flag that tells the loop which system to use.

# [Design Considerations](http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC6)

## [*Assumptions and Dependencies*](http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC7)

The only dependency that affects the system design is the use of GLUT (see SRS document). In addition to 3d graphic capabilities, GLUT also provides you with game loop functionality as well as functions that handle inputs (mouse/keyboard). Therefore, our system design must be compatible with the systems that GLUT has provided for us.

The use of SDL for generating sounds will not require much effort to integrate into our system, and therefore it does not impact the system design.

## [General Constraints](http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC8)

Performance is the main constraint that will affect the design of the play-game system. However, most of the impact will come from specific implementation of sub-systems. The game is expected to run at around 60 frames per second, so it is the individual algorithms being run every frame that impact performance the most. With the general 3d graphics architecture already provided by GLUT, there is not much room for us to design our play-game system architecture in a way that will improve performance.

# [System Architecture](http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC12)

The main source file contains the main() function, which is the entry point of the program. The main() function contains any initialization needed for GLUT to function. Once the initialization is complete it will jump straight into the game loop. The game loop provided by GLUT is the backbone of the system, so our system architecture is designed around it. Our two main subsystems, the menu system and the play-game system, will be accessed through the game loop by setting flags.

If the flag is set to “menu” then the menu system portion of the game loop will run. The menu system will be responsible for setting the properties of the objects in the play-game system. The information gathered from the user in the menu system will be used as constructor arguments when instantiating game objects in the play-game system. For example, if the user uses the menu to set the terrain size then this value will be stored. When the play-game system runs for the first time it will instantiate all of its objects, and the terrain object constructor will be passed this terrain size.

If the flag is set to “play” then the play-game system part of the game loop will run. The play-game system has 2 responsibilities. First, it must instantiate and manage all game objects that are used by the play-game system. Game objects will not have access to each other, so it is up to the play-game system to know what information needs to be passed between objects. Second, it must control all of the game logic. This includes everything from the general turn-based nature of the game to the specifics of object interaction such as collision detection.

## [*Subsystem Architecture*](http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC13)

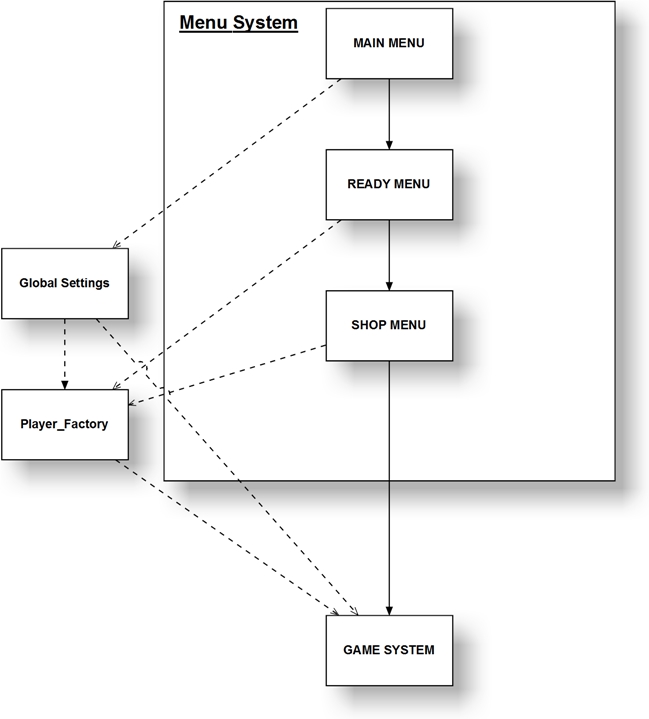
The menu-system design consists of objects such as menus, sub-menus, and control items. Control items are comprised of text input boxes, buttons, slider bars, quantity selection boxes, and checkboxes. The overall architecture of the “main menu” is a collection of control items which consists of a set of “main menu buttons” which point to a given sub-menu. When the mouse is over a “main menu button” and it is pressed and release over the same “main menu button”, its corresponding sub-menu appears in a sub-section of the main menu. Inside each sub-menu you can specify the desired settings during game play. After the desired settings are set the “main player” can click the start game button which launches you into a new “main menu” called the “ready menu”. Inside the ready menu, players select their player type. In the case that a non-AI based player type is selected; they will have the ability to select their tank of choice and set their player name. After each playable character has selected his or her tank, they are directed toward the “shop menu”, where each playable character selects his/her items and weapons.

The play-game system is a combination of all in-game logic as well as several game objects. The terrain object is responsible for generating, storing, and drawing the terrain that the players will play on. The player object is responsible for drawing the player's tank, responding to player controls, as well as managing its own properties such as available weapons (through inventory system), the current position/orientation of the player's tank, and the current hit-points of the tank. The projectile object will draw a projectile fired by a tank, and will track and update the position according to the laws of physics. The skybox object will simply draw a large hollow cube to enclose the play area, and will draw textures on the 6 surfaces to give the illusion of a background. The game system is responsible for passing all relevant data between game objects.

# [Detailed System/Subsystem Design](http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC15)

*Menu System*

To summarize, the overall objective of the menu system, can be seen in the diagram on the next page. The concept behind the diagram is that the main menu system would allow the user to specify his/her settings during the menu system, and that information would then be passed into the game system through a data structure/database.



Overview: The diagram above depicts, at a high-level/detailed description, the interaction between the “Menu System” and the “data structures” used in the “Game System”. After the user has specified his/her settings, and set the number of players and rounds in the “Main Menu”. The “Main Menu” passes this information into a “global settings” object where it is accessed by other objects as the user progresses through the states of the game. Next, when the user hits the “start game button” in the “Main Menu” they move onto the “Ready Menu”. Inside the “Ready Menu”, the players select their tanks (if they are non-AI-based). The information specified per player is then passed into the player factory where it is stored until the user returns back to the main menu. After the “Ready Menu”, each non-AI-based player purchases his/her items to be used in the “Game System”. This information is also passed into the “Player\_Factory” object, which maintains the data for each player (non-AI-based and AI-based). After each of the three menus are completed the game instantiates an instance of a “Game System” which is discussed in the next section.

Input handling: GLUT provides listener functions to detect keyboard and mouse input. When an input is detected, any code within the listener function is executed. All interactions with the “Menu System” are done through the mouse. Each object in the main menu system uses geometry to determine if the mouse was pressed inside its borders. Additionally the mouse handler functions check to make sure that when the mouse is released, it was released over the same Control Item that was pressed previously.

*Play-Game System*

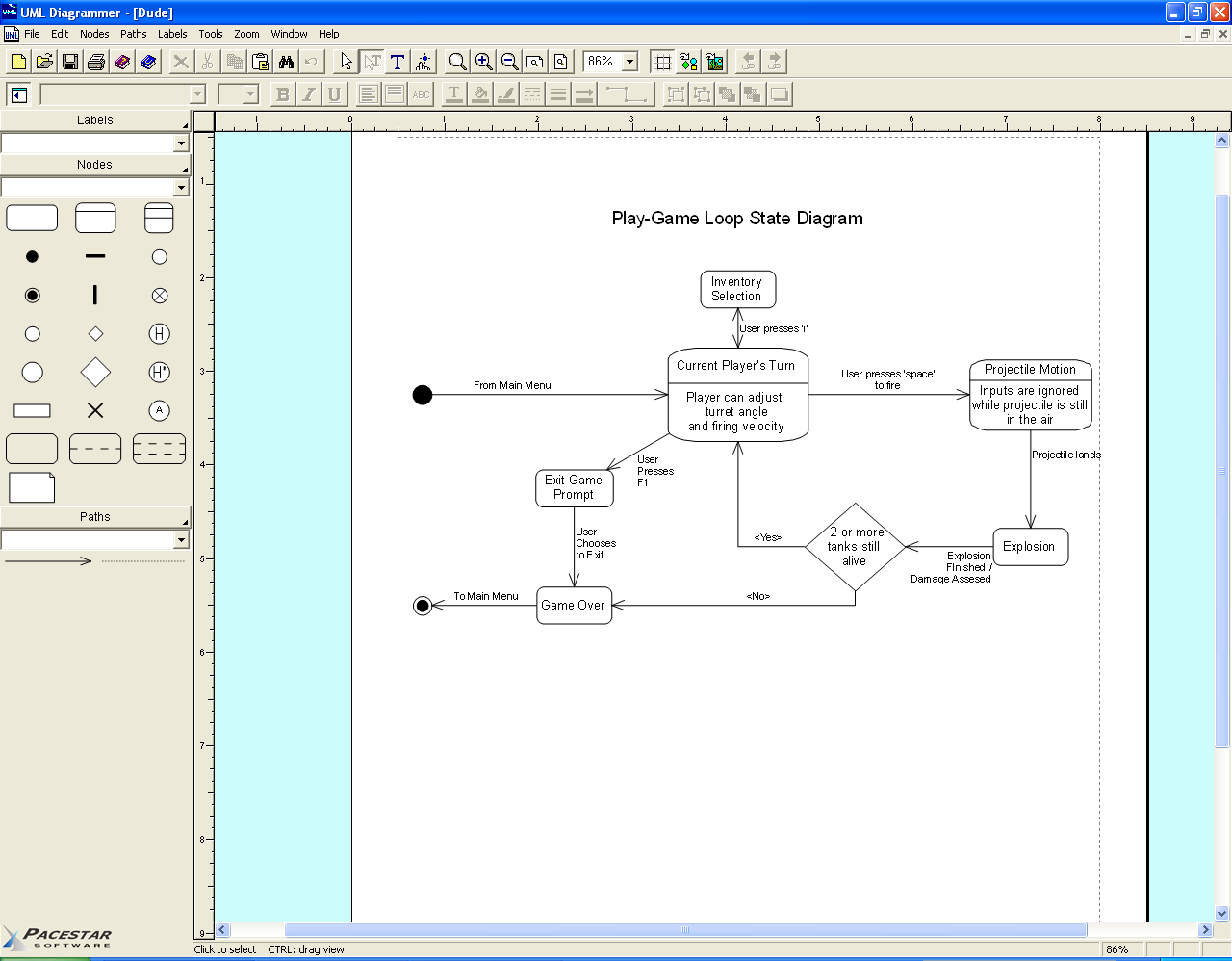
Game logic:

The following elements of game logic will be implemented as functions inside the game loop.

Initialization: This function will instantiate all necessary game objects (Players, Terrain, Skybox) or global variables. Once all initialization is complete a flag will be set to ensure initialization only occurs once at the start.

Input handling: GLUT provides listener functions to detect keyboard and mouse input. When an input is detected, any code within the listener function is executed. Each key on the keyboard is assigned a value which is passed to the listener function so you can determine which key was pressed. Depending on the key pressed, the appropriate action will occur. For example, if a player presses the up-arrow then the keyboard listener function executes. The key value is tested and identified as the up-arrow. The input handler will then call the Player class pitchUp function to rotate the turret up a small amount for that frame.

Turn-based: There will be a global variable (int) for the number of players present and another global (int) used as an index for the current player. During their turn, players can rotate their turret, adjust the firing velocity, access their inventory, fire their weapon, or quit the game. Once a player has used up their turn by firing, the current player index will increment. If the next player has already been destroyed then that player will be skipped. The current player index will wrap around (or use modulo arithmetic) after the last player had his turn. The current player index is used to determine which of the Player objects is currently being controlled by player inputs. If there is only 1 player left alive then the game is over. The following state diagram illustrates the turn-based gameplay: (see diagram below).



Terrain Collision: The play-game system can ask the Terrain object for the height value at a certain coordinate. Tanks will use this data to know what height to be displayed at (so they “sit” on the terrain). Projectiles will use this data to detect when they have impacted with the ground.

Hit Detection and Damage: Similar to collision, the play-game system will keep track of all tank positions and all projectile positions. If a projectile position is within a certain radius of a tank position, then a hit has occurred on a tank. The damage done to the tank is calculated based on the distance from the impact point.

Terrain:

Terrain is a class responsible for storing, generating, smoothing, deforming and drawing the terrain using the following functions and variables:

Storage: The terrain is represented as a large square grid of vertices. Each vertex is assigned a height value. These height values are stored as a 2-dimensional square matrix of floats in which the index of the matrix corresponds to a specific vertex (see below).



Random Terrain Generation: This function is called once at the beginning of the game to randomly generate the terrain. The algorithm currently used is the Brownian algorithm for random terrain generation, but other algorithms may be tested. The Brownian algorithm centers on a vertex and raises all vertices surrounding it by x amount. It then, randomly, either jumps to an adjacent vertex or jumps to a random vertex and raises all surrounding vertices by x amount. This process is repeated y amount of steps. The result is a randomized height map that, given the right constraints, resembles terrain. The values are placed into the 2-dimensional terrain height matrix.

Terrain Smoothing: The terrain produced by the random terrain generator will most likely have some jagged edges or other artificial looking features. In order to make it look natural a smoothing algorithm is run over the terrain height map. The algorithm used is known as a “box blur” and is very simple. It just selects a vertex, takes the average height of all surrounding vertices, and applies that value to the vertex. This repeats for every vertex.

Terrain Deformation: When a projectile lands on the terrain, a crater should be made. When a projectile is fired, the game logic determines where the projectile is when it collides with something. If that something is terrain, then it sends the impact coordinates to the Terrain object which then produces a crater according to the size of the explosion. The deformation is done by simply lowering the values in the terrain height map where the impact occurred.

Terrain Drawing: This function will be called every frame and will display the terrain. It takes the terrain data from the 2-d matrix and draws out a strip of triangles for each row pair. After iterating through all row pairs, the full terrain has been completely drawn onto the screen.

Player:

This is a class that represents a player's tank. It is responsible for managing and updating the position/orientation of a tank, drawing the tank, and providing a camera for the player to use.

Draw: The draw function will draw the tank at it's current position and orientation. The team will create our own tank models and import them into the game to be drawn by this function. The body of the tank must be modeled separately from the turret as the user generally only directly controls the turret.

Position/Orientation: The position and orientation of the tank is represented in 3d space by a 4 by 4 frame matrix. The frame matrix can be divided into 4 components: right vector, up vector, at vector, and a position point. This 3d representation is necessary for OpenGL to display objects correctly. OpenGL also requires this “4x4 matrix” to be implemented as a 1d array with 16 elements (collumn major).

Updates: When the game loop detects an input, the handler will send a message to the current player. The Player class will then take the appropriate action. Functions will be created for each of the actions that the player's tank could take. These include the yaw and pitch of the turret, as well as the initial velocity of the projectile it fires. Yaw and pitch is achieved by using GLUT's glRotate() function which will update the object's frame matrix appropriately. For example, if the game loop detects that the player presses the right-arrow key, then it will call the Player classes' rotateTurretYaw(x) function (which would call glRotate() among other things) which will rotate the turret some arbitrary degrees to the right (this value will be tested for to find the appropriate rotation amount per frame).

Camera: The Player class must provide the user with a movable camera to view the world from. The current plan is to create two cameras. One that directly faces the current player's tank and can only orbit the current player's tank. The other will be a free floating camera to allow players to inspect areas of the world not in line-of-sight. Cameras are implemented with gluLookAt function provided by GLUT.

Inventory: Each Player will have an inventory associated with it. This inventory determines what projectiles are available for the current player to fire. The inventory is represented as a 1d array of integers which is passed to the player constructor when the player objects are instantiated. The index of the array corresponds to a specific weapon, while the value is the amount of ammunition the player has for that weapon. For example, inventorySlot[0] could represent the 'baby missile' while inventorySlot[1] would represent the 'large missile'. A player can only select a specific weapon type as long as the integer value in the array for that specific weapon is .

Projectile:

This is a class that manages any projectile fired by a player. It will draw the projectile, as well as manage and update the projectile's position based on real-world physics.

Draw: Currently this is just a sphere, which can be easily drawn with GLUT. The size of the sphere would represent how powerful the projectile is. This may be replaced with projectile models created by the team.

Position/Orientation: The position and orientation of the projectile is managed in the same way as for the player, with a frame matrix. The initial position/orientation will be the same as the tank that fired it.

Current Velocity: The current velocity of a projectile will be stored as a 3d vector. The initial value is determined by the tank that fired it.

Updates: When the projectile is created, its constructor will receive the gravitational magnitude and a 2d wind vector. Gravity directly affects the 'y' component of the current velocity vector, while the wind affects the 'x' and 'z' components of the current velocity vector. This is done on every frame in which the projectile is active.

Skybox:

This is a simple class that adds a textured background to the environment. It must draw itself and load/manage a texture file.

Draw: The background will be made by drawing 6 squares in a cube pattern, in such a way that it encloses the visible environment.

Texture: Each of the 6 faces must have a texture applied. The textures will be created by the team. The textures must have smooth edges so the transition from one surface to an adjacent surface is not noticeable.