**Ghost Hunter X**

Tech Design Document

Contents

[1 Overview 3](#_Toc343197593)

[2.1 Naming Conventions 3](#_Toc343197594)

[2.1.1 Class Names 4](#_Toc343197595)

[2.1.2 Class Attribute Names 4](#_Toc343197596)

[2.1.3 Method and Function Names 4](#_Toc343197597)

[2.1.4 Method Argument Names 4](#_Toc343197598)

[2.1.5 Include Units in Names 4](#_Toc343197599)

[2.1.6 Variable Names on the Stack 4](#_Toc343197600)

[2.1.7 Pointer Variables 5](#_Toc343197601)

[2.1.8 Global Variables 5](#_Toc343197602)

[2.1.9 Static Variables 5](#_Toc343197603)

[2.1.10 Enum Names 5](#_Toc343197604)

[2.1.11 Macro and #Define Names 5](#_Toc343197605)

[2.2 Comments 5](#_Toc343197606)

[2.3 Minimize Dependencies with Abstract Base Classes 6](#_Toc343197607)

[2.4 Code Length of Methods 6](#_Toc343197608)

[2.5 A Line of Code Should Not Exceed 130 Characters 6](#_Toc343197609)

[2.6 Brace formatting and If Then Else Formatting 6](#_Toc343197610)

[2.6.1 Condition Formatting 7](#_Toc343197611)

[2.7 Code Reviews 7](#_Toc343197612)

[3 Integration Plan 8](#_Toc343197613)

[4 Milestone Deliverables 8](#_Toc343197614)

[4.1 Proof of concept 8](#_Toc343197615)

[4.2 Pre-Alpha Phase 8](#_Toc343197616)

[4.3 Alpha Testing Phase 8](#_Toc343197617)

[4.4 Beta Testing Phase 8](#_Toc343197618)

[4.5 Release Phase: Gold Version 9](#_Toc343197619)

[6 System Architecture 9](#_Toc343197620)

[7 Testing Plan 9](#_Toc343197621)

[7.1 Internal 9](#_Toc343197622)

[7.2 External 9](#_Toc343197623)

[8 Module Breakdowns 10](#_Toc343197624)

[8.1 Render Module 10](#_Toc343197625)

[8.2 Sound Module 11](#_Toc343197626)

[8.3 Artificial Intelligence 12](#_Toc343197627)

[8.3.1 Player AI 12](#_Toc343197628)

[8.3.2 Pathfinding 14](#_Toc343197629)

[8.4 Input 16](#_Toc343197630)

[8.5 Asset Manager 18](#_Toc343197631)

# 1 Overview

The purpose of this document is to create a standardized approach for all members to adhere to during the development of Ghost Hunter X. This document will:

* Detail the coding standards that all group members will follow in order to keep the code consistent and understandable by all members.
* Detail the code integration plan.
* Establish a development plan for the order of and contents of each module which will be constructed.
* Provide a description of what must be met at each milestone.
* Describe the system architecture of the game engine2 Coding Standards

## 2.1 Naming Conventions

* Class, method, and variable names will use Apps Hungarian notation.
* Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough.
* Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design.
* Avoid the temptation of bringing the name of the class a class derives from into the derived class's name. A class should stand on its own. It doesn't matter what it derives from.

### 2.1.1 Class Names

* Use upper case letters as word separators, lower case for the rest of a word
* First character in a name is upper case
* No underbars ('\_') (underbar is only acceptable if it is “\_p” to represent a pointer)
* **Don't put more than one class in a file**. Otherwise, how will I know it’s there when I’m looking for the .h or cpp

### 2.1.2 Class Attribute Names

* use the same rules as for class names.

### 2.1.3 Method and Function Names

* Usually every method and function performs an action, so the name should make clear what it does: CheckForErrors() instead of ErrorCheck(), DumpDataToFile() instead of DataFile(). This will also make functions and data objects more distinguishable.
* Classes are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.
* Use prefixes like “Is” if the method is querying something. Readers will immediately know it’s a question

### 2.1.4 Method Argument Names

* The first character should be lower case.
* All word beginnings after the first letter should be upper case as with class names.

### 2.1.5 Include Units in Names

* If a variable represents time, weight, or some other unit then include the unit in the name so developers can more easily spot problems. For example:
* uint32 mTimeoutSecs;
* uint32 mPlayerWeightLbs;

### 2.1.6 Variable Names on the Stack

* use all lower case letters
* use '\_' as the word separator. For example:
* Time\* out\_of\_time = 0;
* With this approach the scope of the variable is clear in the code.

### 2.1.7 Pointer Variables

* pointers should be prepended by a \_p'' in most cases
* place the \* close to the pointer type not the variable name. For example:
* Renderer\* renderer;

### 2.1.8 Global Variables

* Global variables should be prepended with a 'g'. For example:
* gAudioManager\* audioManager;

### 2.1.9 Static Variables

* Static variables may be prepended with 's'.
* Its important to know the scope of the variable
* For example:

private:

static StatusInfo sStatus;

### 2.1.10 Enum Names

* Labels All Upper Case with '\_' Word Separators. For example:
* AI\_PATROLLING

### 2.1.11 Macro and #Define Names

* Put #defines and macros in all upper using '\_' separators.
* This makes it very clear that the value is not alterable

## 2.2 Comments

* Consider your comments a report describing the system. Expect your comments to be read by someone who has no idea what this code does. Class comments are one part of the report, method arguments another part, and method implementation another part. All these parts should weave together and inform someone else at another point in time just exactly what you did and why.
* Add a comment answering any questions you may come across as you are writing your class/variables/methods. Chances are someone else will have the same question if they look at your code.
* Write comments as you code. You wont ever go back later and document your code.
* So when you do something document it right then and there. When you create a class, document it. When you create a method, document it. And so on. That way when you finish coding you will also be finished documenting.

## 2.3 Minimize Dependencies with Abstract Base Classes

class Jumpable

{

public:

virtual void Jump() = 0;

}

class Frog : public Jumpable

{

public:

virtual void Jump() { ... }

}

etc ...

* The Jump method of each object contains the implementation of jumping behavior for that type of object.
* Gui doesn't have to know what kind of objects are jumping. If they inherit from abstract base class Jumpable, then they can jump.

## 2.4 Code Length of Methods

* Methods should not exceed a single screen of code.
* Consider creating helper functions to reduce the size of methods

## 2.5 A Line of Code Should Not Exceed 130 Characters

* Seriously, no one wants to scroll the screen to the right to read that line

## 2.6 Brace formatting and If Then Else Formatting

if (condition) // Comment

{

Float sum = 0.0f;

while( sum < 50.0f )

{

if(true == donationAdded)

{

for( int i =0; i < donations.size(); i++ )

{

Sum+= donations[i];

}

}

}

else if (condition) // Comment

{

}

else // Comment

{

}

### 2.6.1 Condition Formatting

* The constant value should be on the left of the ==.
* For example:
* If(6 ==runSpeed)
* The above practice will catch errors where some forgets the second = and sets runSpeed = 6.

## 2.7 Code Reviews

* The following 3 paragraphs were taken from a well-thought coding standards website

“First, code reviews are way too late to do much of anything useful. What needs reviewing are requirements and design. This is where you will get more bang for the buck.”

“Get all relevant people in a room. Lock them in. Go over the class design and requirements until the former is good and the latter is being met. Having all the relevant people in the room makes this process a deep fruitful one as questions can be immediately answered and issues immediately explored. Usually only a couple of such meetings are necessary.”

If the above process is done well coding will take care of itself. If you find problems in the code review the best you can usually do is a rewrite after someone has sunk a ton of time and effort into making the code "work." (<http://www.possibility.com/Cpp/CppCodingStandard.html#codereview>

# 3 Integration Plan

Our team will be using version control software called SmartGit to handle our integration process. Each member will create their own working branch where they will implement their assigned coding tasks each week. Members are responsible for pulling the latest changes from the master branch to receive the updates from other members. Members must commit their changes with a detailed comment describing what they have added to the game engine. Then, members must submit a push request to the master branch once their assigned code task is complete and without errors. The team leader will review all push requests and ensure that no one has submitted broken code to the master branch.

In the event that integrated code breaks the main build, it will be the responsibility of the author of the broken code to repair their functions and re-integrate the newly fixed code into the main build to get it working again. The likelihood of this occurrence will be minimized by effective code review. Additionally, each working build will be archived and stored on several thumb drives in the event of the version control server being unavailable.

# 4 Milestone Deliverables

## 4.1 Proof of concept

1. Rendering Engine draws level, and animates player and enemy.
2. AI module controls an enemy’s state machine
3. Physics engine will implement collision between everything
4. Player’s core mechanics are prototyped

## 4.2 Pre-Alpha Phase

1. Controllable player
2. At least one enemy type that is fully controlled by AI and fights the player
3. The player and enemy are animated
4. At least one level demonstrating the core game mechanics

## 4.3 Alpha Testing Phase

1. All of player’s game mechanics are implemented
2. At least two enemy types are fully controlled by AI and fight the player differently
3. The player and enemies are animated and using the intended final art assets.
4. At least one level demonstrating all of player’s game mechanics
5. Menu system is complete with options menu, pause menu, and a list of controls page.

## 4.4 Beta Testing Phase

1. Beta testing and feedback forms.
2. Discuss feedback and implement changes.
3. Categorize listed bugs into level of impact.
4. Fix as many bugs as possible.
5. Menu system has all final art and sound assets and is fully complete.

## 4.5 Release Phase: Gold Version

1. All feedback changes implemented
2. Existing bugs are virtually undetectable or non-critical
3. Polish as much as possible

# 6 System Architecture

The overall architecture of the game engine is to have separate modules that will be functional-based and loosely coupled. The systems and modules that govern the execution of the game will be driven by function calls. These function calls will be designed to have minimal dependencies. Functions will be reviewed during Code Reviews to ensure that they are not overly dependent on other functions and modules to perform their intended tasks. This will allow for easier bug tracking.

Each game module will make use of the singleton pattern to initialize and point to the proper module when functions need to use that module. The modules that make up our game engine are the Rendering module, Havok physics module, AI module, AudioManager, Lua Scripting module, and User Input module.

# 7 Testing Plan

## 7.1 Internal

Internal testing of newly-implemented features will occur during the next code review. In addition, team members will sit down at least once a week to run the game and test as many parts of the code as they can. Members will try to break things and document any bugs that arise and analyze what needs to be changed to fix these issues.

## 7.2 External

External testing will be performed during the alpha and beta phases of development. During each phase, the development team will observe the testers play the game without giving the testers any feedback. Testers will fill out a questionnaire about the game so the development team can go over documented feedback. The questionnaire will be written prior to each phase of testing.

During the Alpha phase of testing the game may not look great, but all features that are intended in the final version should be in the game.

During the Beta phase of testing Art and Sound should be finalized, but modifiable based on feedback. The game should be finished and ready for polish. The beta test will focus more on game play, to see if testers find the game fun. The questionnaire will also be checking if there are any balance issues or other tester feedback. The development team will discuss beta feedback and determine what features to modify, polish, or remove before gold release.

# 8 Module Breakdowns

## 8.1 Render Module

**Render Class**

The render class is in charge of displaying 2D sprites and 3D models. It will convert GUIDs through the resource manager in order to obtain a file’s path and display it.

**Members**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| \_vdbEnabled | Bool | Check for Visual debugger for havok |
| \_context | hkpPhysicsContext\* | The physics content for havok |
| \_vdb | hkVisualDebugger | The havoc visual debugger |
| \_jobQueue | hkJobQueue\* | The havok job que |
| \_threadPool | hkJobThreadPool\* | Havok’s thread pool |
| \_pWorld | hkpWorld\* | Havok’s physics world |
| \_pWorldInfo | hkpWorldCinfo | Info about global simulation parameters |
| p\_Mesh | ID3DXMesh\* | A pointer to a mesh for havok |
| textures | std::vector<IDirect3DTexture9\*> | A vector list for textures |
| materials | std::vector<D3DXMATERIAL> | A vector list for materials |
| numMaterials | DWORD | A handle for number of materials |
| position | D3DXVECTOR3 | A vector3 for position to use |
| rotation | D3DXVECTOR3 | A vector3 for rotation to use |
| scale | D3DXVECTOR3 | A vector3 for scale to use |
| world | D3DXMATRIX | A matrix for the world |
| name | String | Holds the name for an item |
| itemAmount | Int | Holds the quantity of the item |
| gotItem | Bool | Checks to see if player got item |
| renderItem | Bool | Checks to see if item or item list needs to be rendered |
| itemAmountReceived[] | Int array | Assigns a quantity for item to have stored |
| Position | D3DXVECTOR4 | Holds objects position |
| Scale | D3DXVECTOR3 | Used to scale the object |
| Health | Int | Holds health for object |
| isAlive | Bool | Checks if object is alive |
| objectMesh | Mesh\* | Holds a pointer to objects mesh |
| Shape | Short | Stores the type of shape havoc uses |
| bodyInfo | hkRigidBodyCinfo | Havoks rigid body info |
| Mass | hkReal | Holds the mass for the object for havoc |
| shapeSize | hkVector4 | Holds the shape size for havoc for an object |
| D3Dpp | D3DPRESENT\_PARAMETERS | Used to set up window they way we want it |
| DevType | D3DDEVTYPE | Stores the information relating to the computer hardware configuration |
| RequestedVP | DWORD | Stores the type of Vertex Processing we want to use |
| D3DLight | D3DLIGHT9 | Stores information about a light |

**Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| Return | Name | Parameters | Description |
| void | initHavok() | void | Initializes Havok |
| void | initMemory() | void | Finds and sets up memory space for havoc to run |
| bool | initPhysicsWorld() | void | Creates a physics world for havoc |
| void | initVDB() | void | Initializes the visual debugger for havoc |
| void | deinitHavok | void | Deinitialized havoc |
| void | deinitVDB() | void | Deinitializes the havoc debugger |
| void | beginSprite | void | Starts 2d render |
| void | endSprite | void | Ends 2d render |
| void | endRender | void | Ends render |
| Void | stepPhysicsSimulation() | Float dt | Puts both havoc physics and visual debugger into one update |
| Void | HavokCore() | bool vdbEnabled | This sets up visual debugger |
| Mesh | Mesh() | Vois | Constructor for mesh |
| Void | computeWorld() | void | Computes the world for the mesh |
| Dword | GetNumMaterial() | Void | Gets the number of materials from textures |
| Void | Update | Float delataTime | Updates the object |
| Void | convertPosition() | hkVector4\* phyPosition, D3DXVECTOR4\* m\_Position | This converts the object position to the world |
| Void | creatHavokObject() | hkpWorld\* world | This creates a havoc object |
| Void | setItemName() | String \*s | Sets the items name |
| String | getItemName() |  | This gets the items name |
| Bool | PlayerGotItem() | Bool gotItem | This checks to see if player got item |
| Void | AddItemToArray() | Bool gotItem | Adds item to array if player got an item |
| Bool | RenderItem() | Bool renderItem | This checks to see if item needs to be rendered` |
| HWND | M\_hWnd |  | Handle to window |
|  |  |  |  |

## 8.2 Sound Module

**Sound Class**

The sound module handles the loading, playing and changing attributes of game sounds and music. The update function is entirely for FMOD purposes and contains no information or methods that alter the sounds. To modify their attributes, the set functions must be used from outside the class. For layering, two music files will be played at the same time, one of which will have an initial volume of zero. When necessary, the layered file will steadily increase in volume, to produce the layering effect (e: for intense battles).

**Members**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| (\*)pSystem | FMOD::System | FMOD System variable |
| (\*\*)pMusic | FMOD::Sound | Music track number |
| (\*\*)pSound | FMOD::Sound | Sound effect number |
| (\*\*)pChannel | FMOD::Channel | Channel variable |
| currentTrack | short | Current track, for music only |

**Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| Return | Name | Parameters | Description |
| void | init | void | Initializes the FMOD sound system and sets default values |
| void | setVolume | std::string track, float vol | Changes a channel’s (track) volume |
| void | playSound | std::string track, float vol | Plays a sound based on its GUID (a string) |
| void | playMusic | std::string track, float vol | Plays a track based on its GUID (a string) |
| void | stopMusic | std::string track | Stops a track |
| void | update | void | Runs FMOD’s system update |
| Void | loudSounds | void | Loads all game sounds and music |
| void | shutdown | void | Releases the module |

## 8.3 Artificial Intelligence

### 8.3.1 Player AI

Description: cPlayerIntel – PlayerIntel will represent a challenging and satisfactory opponent for single-player games. It will use as many player accessible functions as feasible, while storing map discovery information in a log that will update only when discoveries are made. The log will be the driving force of prioritizing the AI player’s objectives.

**Members**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | | | Description |
| MaxCurrentUnits | int | | | Maximum Units Allowed |
| UnitCount | Int | | | Count of Current Units |
| NumOfSpheres | Int | | | Count of Current Spheres |
| NumOfCubes | Int | | | Count of Current Cubes |
| NumOfCylinders | Int | | | Count of Current Cylinders |
| NumOfPyramids | int | | | Count of Current Pyramids |
| ObjectivePriority | | int | Value of Prioritization | |
| WorkshopRand | | int | Random value sent to WS | |
| NodeCapCount | | int | Count of Nodes Captured | |
| NodeDiscCount | | int | Count of Nodes Discovered | |
| TowerTotal | | int | Count of Active Towers | |
| AdvMeshCount1 | | int | Count of AdvMesh type 1 | |
| AdvMeshCount# | | int | Count of AdvMesh type # (# of Adanced Meshes in final version) | |
| AttackerArray | | Unit[] | Array of Units sent to attack | |
| ExplorerArray | | Unit[] | Array of Units sent to capture | |
|  | |  |  | |
| BuiltAll | | bool | Flag if all production buildings built | |
| Attacked | | bool | Flag if under attack | |
| Attacking | | bool | Flag if attacking | |
| CaptureMode | | bool | Flag for Capture Mode | |
| AttackMode | | bool | Flag for Attack Mode | |
| BuildMode | | bool | Flag for Build Mode | |
|  | |  |  | |
| DiscLogOut | | ofstream | file output stream to Disc Log | |
| DiscLogIn | | ifstream | file input stream to Disc Log | |

**Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| Return | Name | Parameters | Description |
| void | DecisionTree | bool BuiltAll  bool AttackMode  bool BuildMode  bool CaptureMode  bool Attacked  bool Attacking  int MaxCurrentUnits  int UnitCount  int NumOfSpheres  int NumOfCubes  int NumOfCylinders  int NumOfPyramids  int NodeCapCount  int NodeDiscCount  int\*ObjectivePriority | Function that takes in all pertinent information and shifts the Objective Priority based on existing variables. Essentially the Brain of the AI, it will set Modes and and call other functions based on those Modes and other variables. |
| bool | StartBuilding | int MaxCurrentUnits  int UnitCount  int NumOfSpheres  int NumOfCubes  int NumOfCylinders  int NumOfPyramids  int\*ObjectivePriority | Recursive Operation to begin and continue constructing Production Buildings until All buildings have been built. |
| bool | StartCapturing | int NodeCapCount  int NodeDiscCount  Unit[] ExplorerArray  int\*ObjectivePriority | Function to assign a set of units towards finding and capturing Nodes |
| bool | StartDefending | bool Attacked  int UnitCount  Unit[] ExplorerArray  Unit[] AttackerArray  int\*ObjectivePriority | Function to command all exploring and attacking units to respond to enemy attack |

## 8.3.2 Pathfinding

Description: cPathfinding – Path finding will be responsible for finding, storing, and updating all path finding related activity. The path finding will up updated by itself as it calculates the path every 1-2 seconds.

Stipulations: Only the pathfinder will create and update the path, it can be destroyed by other classes depending on what the other class is capable of doing.

**Members**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | | **Description** |
| mapWidth | Int | | The width of the map |
| mapHeight | Int | | The height of the map |
| tileSize | Int | | The size in pixels of each tile |
| onClosedList | Int | | The size of the closed list |
| notFinished | Int | | Path-related constant |
| notStarted | Int | | Path-related constant |
| found | | Int | If path found |
| nonexistent | | Int | If path does not exist |
| walkable | | Int | Is the node walk able |
| unwalkable | | Int | If the node is un-walk able |
| walkability | | char | Creates array of what is walk able |
| openList | | Int | 1d dimensional array holding ID# |
| whichList | | Int | 2d dimensional array used to record if cell is open or closed` |
| openX | | Int | 1d array stores the x location of an item on the open list |
| openY | | Int | 1d array stores the y location of an item on the open list |
| parentX | | Int | 2d array to store the parent of each cell (x) |
| parrentY | | Int | 2d array to store the parent of each cell (y) |
| fCost | | Int | 1d array to store F cost of a cell |
| gCost | | Int | 2d array to store G cost of a cell |
| hCost | | Int | 1d array to store H cost of a cell |
| pathlenght | | Int | stores length of the found path |
| \*pathBank | | Int | Stores paths for each shape |
| pathStatus | | Int | The place in the path each shape is at |
| xPath | | Int | The x location on the current path |
| yPath | | Int | The y location on the current path |

**Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| **Return** | **Name** | **Parameters** | **Description** |
| void | InitializePathfinder | void | Allocates memory |
| void | EndPathfinder | void | Frees memory |
| int | FindPath | Int pathfinderID  Int startingX  Int startingY  Int targetX  Int targetY | Finds a Path using A\* |
| void | ReadPath | Int pathfinderID  Int currentX  Int currentY  Int pixelsPerFrame | Read the path data and convert it to screen pixel coordinates.  \*PixelsPerFrame is used to determine if the pathfinder has gotten close enough to the center of a given path square to warrant looking up the next step on path |
| int | ReadPathX | Int pathfinderID  Int pathlocation | Reads the x coordinate of the next path step |
| int | ReadPathY | Int pathfinderID  Int pathlocation | Reads the y coordinate of the next path step |

## 8.4 Input

Description: cDirectInput – DirectInput will handle all input from keyboard and mouse. DirectInput will only handle the commands from the keyboard and mouse, depending on what is highlighted or targeted will determine what the input will do.

**Members**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| DInput | IDirectInput8\* | Creates the direct input |
| Keyboard | IDirectInputDevice8\* | Used to create the keyboard input |
| KeyboardState[256] | char | Gets the current state of the keyboard |
| Mouse | IDirectInputDevice8\* | Used to create the mouse input |
| MouseState | DIMOUSESTATE2 | Gets the current state of the mouse |
| MousePosition | D3DXVECTOR3 | Vector for mouse location |
| PickRay | D3DXVECTOR3 | Vector for rayCast |
| pInstance | static CInput\* | Used to store a singleton to the input class |
| InputCleanup | static CSingletonCleanup | Used to delete the singleton |

**Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| **Return** | **Name** | **Parameters** | **Description** |
| void | Poll ( update ) | () | polls the keyboard and mouse for current state |
| bool | keyUp | char key | returns KeyboardState |
| bool | keyDown | char key | returns mkeyboardState |
| bool | mouseButtonDown | int button | returns mMousestate |
| float | mouseDX | () | tells change in x position of mouse |
| float | mouseDY | () | tells change in y position of mouse |
| float | mouseDZ | () | tells change in z position of mouse |
| d3dxvector3 | getMousePosition | () | returns mMousePosition |
| float | castRay | d3dxvector3 mousePosition | returns object that it intersected |
| static CInput\* | Instance ( init ) | DWORD keyboardCoopFlags, DWORD mouseCoopFlags, HINSTANCE\* pAppInst, HWND\* hMainWnd | Used to create the singleton instance of CInput |

## 8.5 Asset Manager

Description: cLevelEditor – The level editor will be a tool used for level layout and creation. The editor will allow developers to load in a height map which will produce the hills and valleys of our terrain. From there the developer will be able to strategically place resource nodes, starting buildings, and starting units for that level. The level will then save to a CSL file.

**Members**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| ObjectInfo | Struct: float32[3]  float32[3] int | Struct used to contain position and rotation information in float32[3]s and object type ID in an int. |
| nodePlayer1Start | ObjectInfo | The starting location for the second player and a resource node |
| nodePlayer2Start | ObjectInfo | The starting location for the first player and a resource node |
| nodeEmpty | ObjectInfo | The location of an empty resource node |
| nodeList | SLL of ObjectInfo | Linked list to contain information for the location of each object along with an id number describing what the object is. This list will be written to a file that will be loaded back in for the level. |
| cursor | Float[2] | Location of the mouse cursor |
| currentObject | int | ID for the current object attached to cursor, ready to be placed. |
| buildingCube | ObjectInfo | Location of a cube producing building |
| buildingSphere | ObjectInfo | Location of a sphere producing building |
| buildingCylinder | ObjectInfo | Location of a cylinder producing building |
| buildingCombiner | ObjectInfo | Location of a combination factory building |
| unitSphere | ObjectInfo | Location of a sphere unit. |
| unitCylinder | ObjectInfo | Location of a cylinder unit. |
| unitCube | ObjectInfo | Location of a cube unit. |
| unitPyramid | ObjectInfo | Location of a pyramid unit. |
| terrain | string  float[3] | Name and center location for the heightmapped terrain grid. |

**Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| **Return** | **Name** | **Parameters** | **Description** |
| Void | AddObject | Float[3] int  Float[3] | Adds the object to the list of objects for the level when the user places an object. Float[3]s represent the position and rotation while int represents the object type. |
| Void | DeleteObject | Float[3] int  Float[3] | Deletes selected object from list of objects. |
| Void | Export | Float[3] int  Float[3] | Writes a CSL file with all the info for the level. The name for the terrain file will be written first, followed by location, rotation, and ID for all objects. |
| void | Import | string | Load the requested level file. |
| void | LoadTerrain | string | Load the requested terrain from heightmap and convert to terrain trigrid. |