Project Derelict

Team BackLit

GAM 300

Spring 2011

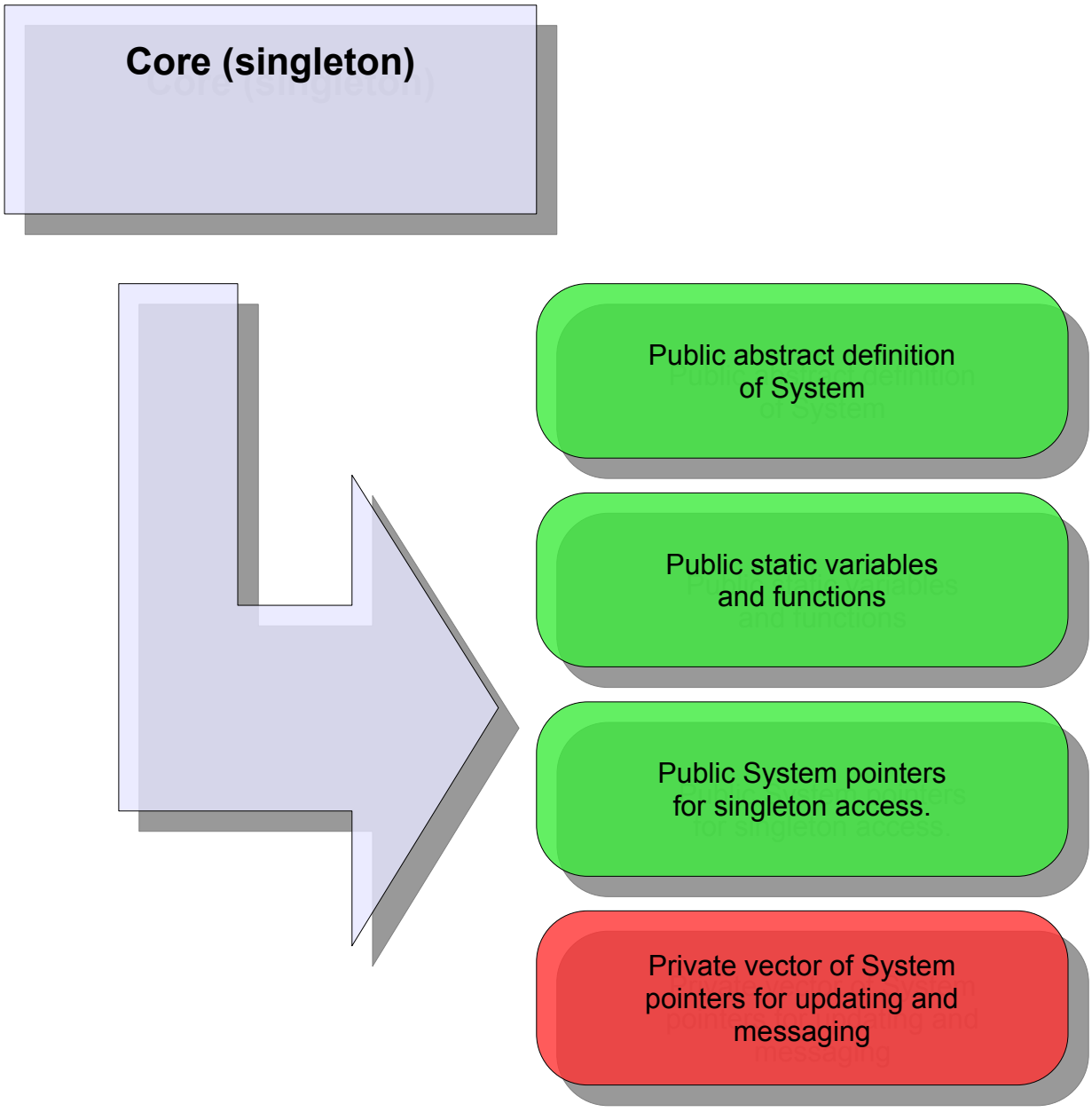
Ryan Davey – Technical Director
Bryan Bishop - Producer
Craig Sutton – Game Designer
Steven Chith – Graphics Programmer
Daniel Bloom – Network Programmer
Branden Gee – Tools, Gameplay, Serialization Programmer
Stephanie Keene – Art Director
Sarah Nixon - Animation
Kayla Oswald – Characters and Environments
Danny Huynh – Characters and Environments
Jenn Ryckman – Characters and Environments
Jonathan Sokol – Biofeedback Engineer

Table of Contents

Overview

The Engine for Derelict is built from the ground up using only cross-platform APIs for Graphics, Audio, and other systems. The engine builds and runs on Windows, Linux, and Mac OSX.

Core Overview



The Engine's Core is a singleton object which provides the main entry point into the engine. It contains 4 main points of functionality.

- **Public abstract definition of System:** This is the abstract definition of a System object, which are the main modules used to make up the game.
- **Public static variables and functions:** Variables and functions defined to be static can be accessed by any object which is a System. These are comprised of variables such as accumulated time and frame rate, and functions for allocating and de-allocating memory, as well as a function for broadcasting messages.
- **Public System pointers for singleton access:** Pointers to systems may be added as public data that can be accessed globally due to the Core being a singleton. This is mainly for script integration purposes.
- **Private vector of system pointers for updating and messaging:** A vector of System pointers are contained privately within the core. The core loops through these pointers to initialize, update, and destroy the systems. The order of the systems is determined what order they are added to the Core.

Systems Overview

A system is a modules which provides a specific, main block of functionality to the engine (i.e. Graphics, Physics, Networking, etc.) This engine will use thirteen such systems. Components and game objects (component compositions) are managed by one of these systems. For more information on the component and game object architecture, see Object Management.

Audio

The Audio system uses FMOD to manage audio data and functionality. The system will provide components that will give game objects audio functionality. It will also provide a way for playing sounds using messages. The system will use the FMOD API to perform post processing effects on certain sounds as well as perform three dimensional audio.

Biometric

The Biometric system is the main interface for connecting our custom biometric device. The system handles interfacing with our custom hardware peripheral (see Heart rate Monitor), reads data from it, and provides that data to the Engine.

Game Logic

The Game Logic System houses all of the general callbacks into the Javascript scripting engine which are not specifically related to other systems. This system also manages the initialization specifics of the scripting engine (see Javascript) as well as other management functionality related to scripting.

Graphics

Graphics is implemented using OpenGL through SDL version 1.3 and GLEW 7.0. SDL is being used in unison with SDL_Image, which provides texture loading and processing functionality to the Engine. The graphics system uses FreeType version 2.4.6 for text rendering. For more information about the Graphics system, see Graphics Implementation.

Input

Input utilizes SDL to intercept input data from the operating system which is cross-platform fashion. Once intercepted, the messages are pushed out to the engine as messages (see Messaging). This system also handles input events for AntTweakbar (see Debugging.)

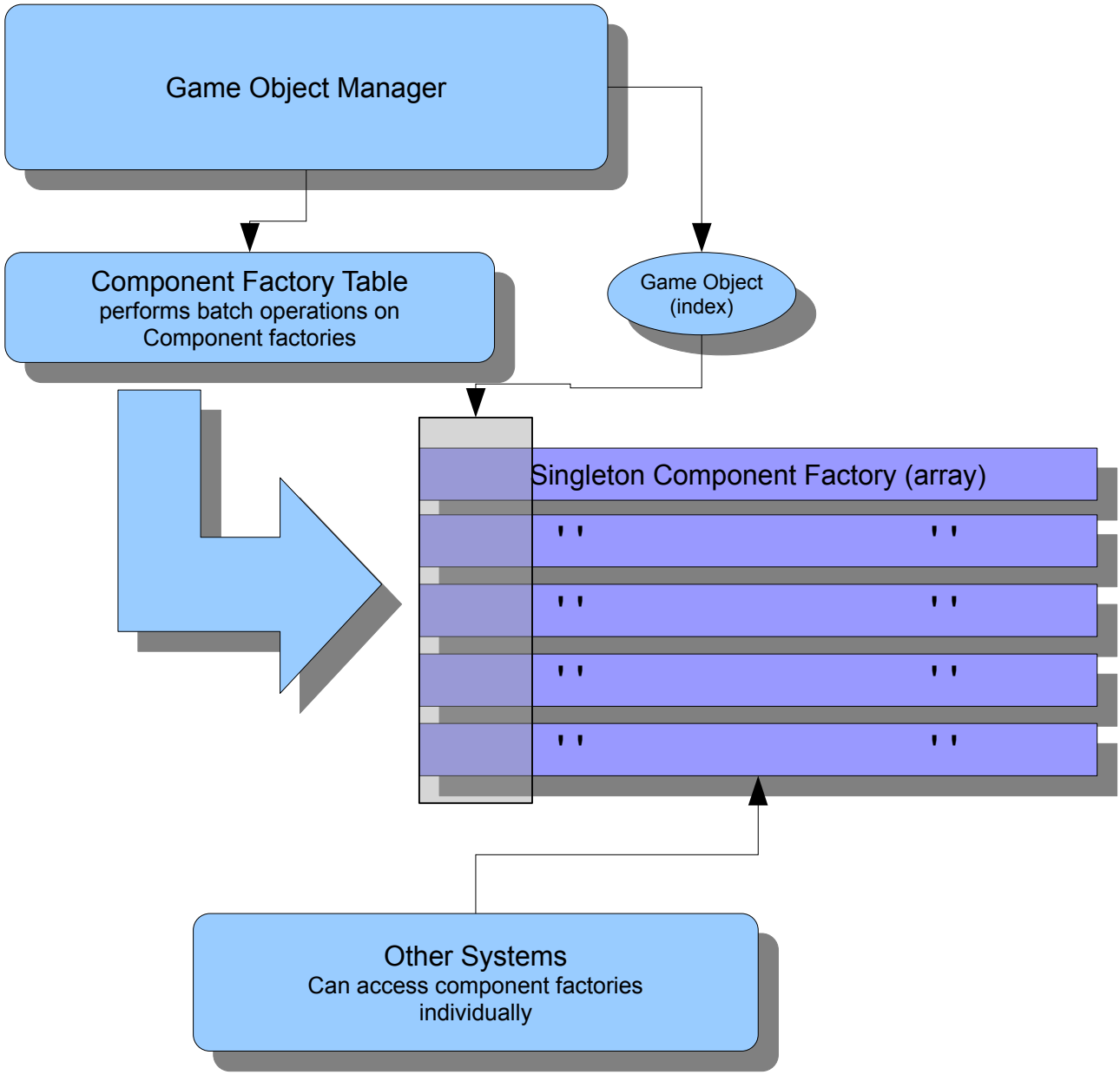
Memory Management

The memory manager system is accessible through the Core. It is a collection of separate object allocation modules that can be accessed in constant time. It provides allocation of memory in blocks of different sizes. It also provides STL list, map, set, mulimap, and multiset containers that are hooked into the memory manager using a custom allocator in a manner that is clean and generic. This system also provides extensive customization of the sizes of the separate block allocators, the page sizes of those allocators. The system also provides detailed debug output of the state of the object allocators.

Messaging

The messaging system maintains a queue of messages that are pushed out to the engine at the end of each frame. The queues are received by the Core, which provides the BroadcastMessage function to the Systems and Components in the engine. Messages are memory managed, and are all allocated with the same Object Allocator within the memory manager. The messages are passed locally via reference counting smart pointers, which are also thread safe, so the other modules in the engine can cache and manage messages however they like. This is particularly important for the Networking System.

Object Management



The object manager manages the objects in the engine using a handle/ID system. The system provides basic functionality, through a Component Factory Table to Javascript which then manages creating and destroying objects, as well as adding and removing components from these objects. Components are held in Component Factories. Each component type has a component factory that contains space a component for each game object currently in the game.

In this way, Component access and management is very efficient, since game objects simply exist as an index into the component factories (which are parallel arrays) at the cost of having some stale memory. The object management system is set up in a way that allows us to optimize the memory usage if required. The component factories are singletons which can be accessed engine-wide, so that Systems that deal with a specific type of component can easily access and modify them.

Networking

The networking engine is connection-based using a protocol built to run over UDP. Packets are laid out as follows:

Packet Layout

Bit offset	0–7	8–15	16–23	24–31
0	Protocol ID			
32	Hash			
64				
96	ACK bits 0-31			
128	Sequence #	Acknowledgment #		Message Count
160+	Data			

The Protocol Identifier is "DRLT" and the acknowledgment number indicates the last packet received. ACK bit 0 indicates if the previous packet was received, and bit 1 indicates if the packet prior to bit 0 was received.

Message Count indicates how many messages need to be extracted from the packet. Data is composed of messages, each of which must be prefaced with an integer, representing the message type. Connections are dropped if no packets are received for a 10 second period.

If packets are received out of order as determined by sequence number, they are dealt with differently. If they are determined to be too late, they are disregarded. If they are determined to be too early, they are stalled for a short period of time, at which point they are processed normally, along with all other stalled packets which precede them sequentially. Packets which are dropped are resent as messages attached to other packets.

Connections are stored by ip-port combination. Any packet which is received from an ip-port which does not match an existing connection is disregarded. The current networking system does not, however, handle the creation of said connection. That functionality is left unimplemented so as to remain flexible and agnostic of connection establishing procedures.

Physics

The physics system provides rigid body physics simulation to the Engine. This is mainly provided through a component, BodyComponent, and a sub-classification of that component, Shape. Collision detection and resolution is performed between a collection of primitive geometric shapes, including Plane (half-space), Ray, Sphere, Axis-aligned Box, Oriented box, and Triangle.

Movement is calculated using basic Euler Integration (linear and angular). Broadphase is achieved through the use of a BSP tree built form the environment geometry and cordoned into rooms. For more detailed information about the physics engine, see Physics Implementation.

Thread Management

The thread manager system is a simple abstraction layer around basic multi-threading functionality. A thread can be requested from the system by sending it a message containing a function pointer (the thread function) and the thread parameter (containing data). The System maintains a collection of threads, the number of which is configurable, that will sleep if they are not in use, and will run if they are registered to a function. This system exists to keep multi-threading code easy to use and clean, as well as manageable and easy to debug.

Window Management

The window manager system is a basic system that handles the window for the game. It uses the SDL library to create and manage the window. There are features in the system to change the size of the window, as well as change from full screen mode to windowed mode. This system also manages swapping the frame buffer each frame.

Development Console

The Development console manages the in-game Javascript console. There are two main functions of this system. First, it interfaces with the Javascript engine (see Javascript) to execute lines of Javascript typed into the console. Second, it manages printing the text, either received from Javascript, or from the engine, onto the screen for viewing. The Development console also prints any debug output that is processed by the engine.

Graphics Implementation

The Graphics uses OpenGL for rendering models and textures. Textures are converted from FBX format to a custom binary format which is loaded and maintained OpenGL structures. Rendering is performed with a deferred shading method. Shadows are rendered using a cube map, and smoothed using Percentage Close Filtering.

FBX Model Format and Model Viewer

FBX format is used for loading and rendering 3D models and animations, however, there is a proprietary model viewer which supports drag-and-drop of an FBX file onto the viewer. The viewer shows the model as it would appear in the engine, as well as converting the .FBX file and exporting it to a custom (.mesh) format to be read by the engine. The binary format is custom structured for the engine, and as a result, is very efficient.

Deferred Shading

The shading model used in the engine is deferred, 4 pass method with 4 render targets. The render targets are for position, normals, diffuse, and specular. The deferred shader allows for rendering a large number of lights efficiently. The shading pipeline supports point, spot, ambient, and directional lighting, as well as emissive textures.

Asset Management

Internally, the graphics engine manages assets using structures provided in the OpenGL interface. Loading assets happens in parallel to the Engine via multi-threading, and the functionality of the graphics engine is message oriented. The engine communicates with the graphics engine by sending messages, which the graphics engine handles with a custom message queue, and doles out tasks to certain modules of the graphics engine (i.e. a separate thread.)

Behavior Implementation

There will be two types of simple behaviors implemented in Derelict. One is for generating the level in which the players will interact. The second is for generating heart rate and GSR readings for a player that may not have the Biofeedback device.

Level Generation

The level generation is a non-complex algorithm that is designed to place a random set of rooms into a linear formation which will represent the Alien spacecraft. This algorithm is made simple by the strict design rules set in place on the rooms. The rooms can be of a certain maximum size within a bounding cube. They can be of any shape or form within that cube, but there must be two doors for entering and exiting the room, and they must be directly across from each other on the sides of the cube.

The algorithm will essentially be responsible for lining up the rooms in a logical manner according to their type. There will be types of rooms that dictate where in the ship they will be located. For example, end rooms (like cockpit and engine room), and middle rooms.

Heart Rate and GSR Simulation

When one or more of the players does not have a Biofeedback device to read their Galvanic Skin Response and their Heart Rate, an AI system will be used to simulate these readings. The system generates a number that represents your heart rate and GSR depending on a few variables such as the players' proximity to each other, the rate at which they are moving, and/or any other obtainable data that may be pertinent to these readings. A wave form can be easily drawn on the screen based on the number generated by this AI system.

Physics Implementation

The physics that is supported in the engine is of mediocre complexity. It must support zero gravity environments with floating players and objects. Because of this, robust collision resolution such as for stacking is not required.

Movement

Movement is performed using basic Euler integration of linear movement and angular movement. This is applied by forces, which are then integrated using mass and friction. The systems that control movement will have a close relationship with Networking in order to smooth movement for players over the network (see Multiplayer Implementation.)

Collision Detecting and Resolution

Collision detection is performed between a collecction of geometric primitives. These include:

- Spheres
- Axis Aligned Boxes
- Oriented Boxes
- Half-spaces
- Rays
- Triangles

Rays provide a way to perform ray-casting for shooting logic, and provides intersection points with physics bodies. Triangle collision is performed when because geometry is organized into a BSP tree, which allows dynamic objects to collide coherently with static level geometry.

Collision resolution is performed using the separating velocities of the objects about their contact normals, as well as calculating penetration volumes to provide a more robust and accurate collision resolution. Using penetration volumes is also necessary to provide more realistic simulation of rotating objects that collide in zero gravity.

Broadphase

Static level geometry is built into a BSP tree, which is then used to perform collision detection and resolution with the dynamic objects in the world. This tree is also processed into “rooms” using a rooming algorithm. A subsystem within the physics engine manages dynamic objects' states. Objects will sleep if they are not within the immediate room, or adjacent room of the player.

Javascript

The Javascript engine uses V8 to accomplish binding objects and functions to Javascript, serializing and de-serializing using JSON format in Javascript, and to provide an interface for creating game logic using the previous two mentioned features.

Wrapping Objects and Functions

The interface for integrating C++ objects and functions into Javascript is very simple. Objects and functions are wrapped using a combination of descriptive macros. For example, to wrap a class, the class definition must contain jscript_start(ClassName). The variable and function members of the class are wrapped in the same fashion. For Example jscript_declaration(int I) declares a member variable of type int, labeled I.

JSON

Object archetypes are defined with Javascript Object Notation (JSON), in our Javascript engine, specifically in the script source files. There is a system in place within the engine which maintains a library of object archetypes which can be created and initialized at any time in Javascript. In this way, Serialization and De-serialization is performed in conjunction with scripting functionality and is taken care of mostly by the V8 API.

Underlying Implementation

JavaScript wrapped types are structures with a specified number of data members of type v8::Value. CPP types can be converted as well from JavaScript via the overloaded JavaScript::CastToJavaScript and JavaScript::CastFromJavaScript functions. Any type which has been wrapped in the default manner should automatically be convertible to and from JavaScript.

Additionally, any type which defines a member function "v8::Local<v8::Object> GetJScriptRepresentation() const", will also be convertible to JavaScript, although doing so is ill-advised.

The default behavior for user defined types is to wrap a single pointer to the type, with the presumption that the object will be alive for the duration for which it is accessible in JavaScript.

Multiplayer Implementation

For detailed information on the networking engine and protocols used, see Networking. The networking engine handles connections that will support only two players in the game world, since the game is 1v1. There are two methods of seeking connections. One method is through LAN broadcasting, and the other method is through WAN matchmaking. In both cases, the two players will be matched automatically and their game world will be created automatically. Players will be able to choose which side (Alien, or Human) to play on, and matchmaking will happen according to these choices.

The Object Management system provides modes that support a 1v1 style connection and object creation based on being a server or a client. The player is chosen to be either a client or a server automatically based on networking data internal to the engine.

There are no other complex implementations related to multi-player other than the solutions provided by networking. The other logic will be handled by other crucial systems in the engine, and by scripting game logic.

Coding Methods

The members of the team have agreed on a loose set of coding methods to be used in the engine. The major specification relates to cross platform compatibility, in that code written must support both Microsoft's C++ compiler as well as g++. Warning and error messages are maximized on both compilers. Code is written in Microsoft's Visual Studio 2008, and cross platform tests are performed with Code::Blocks, with a relatively current version of the g++ compiler.

Cross-platform

The engine must support cross platform compatibility. Code written must be in compliance with both Microsoft's C++ Compiler as well as G++. This enforces C++ standard code and is a great way to keep implementations across many team members standard and organized.

Coding guidelines

There is a comprehensive set of coding guidelines that are suggested on the team's website. Here is a summary of those guidelines.

Typographic Style

Typographic style pertains to the style of code that a programmer uses. For example, brace placement, parentheses placement, variable and function naming conventions, etc. There are no strict enforcements on the typographic style of coding. You may use whichever style you are comfortable with when writing your own blocks of code, but restricted by the following guidelines:

1. *Your style must be readable, organized and consistent.*
2. *When altering or adding modifications to another teammate's code, you must adhere to their typographic style.*

Tabs

The one pedantic rule that relates to typographic style. You MUST use TABS instead of spaces when coding in the project.

File and Function Headers

Every file in the project must have a file header. Developers must make sure to check these headers occasionally to make sure they are up to date. If you are the major author of a file, but your name is name listed as the author, then feel free to take authorship of the file and list the other person as a contributor. (It is probably a good idea to let this person know about it as well).

There are no specifications for a function header. If you feel that it is necessary, add comments at the top of your function to explain what it does. This is not required, but suggested.

Header Guards

Use the "#pragma once" header guard method at the beginning of every header file.

Basic Types

Developers must use the basic type definitions defined in the SDL library. Specifically SDL_stdinc.h. This is for cross-platform purposes.

Style Suggestions

The team website section for Coding Guidelines provides a number of code examples as style suggestions. This is for programmers on the team (such as designers) who are not sure of what style they want to use.

Bug submission guidelines

The team website has a custom bug tracking system that uses Google Spreadsheets to track and manage Bug, Feature, and Asset tickets. Tickets can be assigned to anyone on the team and requested by anyone on the team. Emails are sent out to pertinent members of the team automatically when tickets are submitted. The spreadsheet can be sorted according to priority, owner, date submitted, and more. There are strict guidelines for submitting Bug type tickets. They are as follows, according to the form submission:

Title

The title of your bug. This is the first thing your teammates will read, so try to explain exactly what the problem is in about one sentence.

Examples:

- *Heap Corruption causing instability in Graphics Engine*
- *Compiler error in WinMain.cpp line 20*
- *Messages are not being deleted somewhere in Network Engine*

Type

Type Bug

Priority

The priority is measured from 1 to 5. 1 being the highest priority, and 5 being the lowest. Priority is directly related to the perceived impact in the stability of the engine due to said bug. For example, Highest priority bugs affect the stability of the engine severely, while low priority bugs affect the stability of the engine very little or not at all.

1. (priority 1)
   - *The bug is directly blocking you from implementing whatever you are currently working on.*
   - *The bug is some kind of memory corruption (heap corruption, stack overflow/underflow, etc.)*
   - *The bug is very common and/or easily reproducible, and severely compromises the stability of the engine.*
   - *The bug is a compiler ERROR that you don't know how to fix. (particularly one that someone else committed.)*
2. (priority 2)
   - *The bug is a memory leak.*
   - *The bug is a Linux or Mac build compiler ERROR.*
   - *The bug is a windows build compiler WARNING (yes, compiler warnings are treated as bugs.)*
   - *The bug is somewhat random, and you don't yet know how to reproduce it.*
   - *The bug is because a member of your team did not commit code or files correctly, and you are not in immediate contact with them.*
3. (priority 3)
   - *The bug is an assert that is being triggered because of code that you, or someone else, wrote, and you don't know how to fix.*
   - *You are unsure about the bug's priority.*
   - *The bug is highly reproducible, and only mildly affects the stability of the engine (i.e. lag, jitter, visual artifacts, collision anomalies, etc.)*
4. (priority 4)
   - *Obvious logical errors in another teammate's code, which may work (but is dangerous), or affects the stability of the engine in a very subtle way. For example, Initialization of variables at incorrect times, reading uninitialized data during the first few frames, etc.*
   - *The bug exists due to an unfinished section of code, or is predicted to not exist once something is finished. Example: BUG: Nothing is drawing on the screen!? Will probably be fixed when the graphics engine is finished. (Note: If you are being blocked from finishing an important feature, you may want to choose priority 1 or 2.*
5. (priority 5)
   - *The bug is speculative, or predicted. For example, you spot something in someone else's code that you might think is logically incorrect, but you are not sure.*
   - *The bug does not affect the stability of the Engine or game. i.e. something "we should probably fix." Be advised that this type of bug should be EXTREMELY RARE. If you think your bug is priority 5 for this reason, you should probably think about it a bit more.*

Assign ticket to

Assign the ticket to whoever you think is responsible for the bug. If you are unsure who is responsible, then you can choose "None." You can also choose "None" if you wish for all of the devs to receive a notification about the bug. If you are assigned a bug which you are not responsible for, you can re-assign to "None" or to whoever you feel is responsible for the bug.

Requester

Whoever you are. (Or someone else, if you're submitting the ticket for someone else.)

Description

This field is for the description of your bug. There are two different forms that you should use for this field depending on the type of bug.

1. **For General Bugs**
    - **Brief**: A brief description of the bug, how it affects stability, what may be causing it, and how it may be blocking you. Feel free to describe and speculate about the bug as much as you want here. More information is always better.
    - **To Repro**: List steps to take to reproduce the bug. If you don't know, state this.
    - **Steps Taken:** List all of the trouble shooting steps you've taken to attempt to find and/or fix the bug.
2. **For Compiler Warnings/Errors**
    - **File Name**: The name of the file that emits the warning or error.
    - **Line Number**: The line number that emits the warning or error.
    - **Warning Message:** The warning message copied and pasted from the compiler output.
    - **Other Info:** Any other info about the warning or error that you wish to provide. (Especially if it is a cryptic or non-intuitive error or warning (i.e. templates))

Comments

If you click on the "View Ticket" button for a certain bug, there is a field for submitting comments about the bug. It is not required, but it is very helpful if you post updates about the process of fixing particularly difficult bugs. Add each new update at the top of the field with the date and time of the update.

For Example: Update 10/4/11 2:18pm : Found the cause of the bug. It happens because memory is being corrupted in a loop in FileName.cpp line 2001.

Additionally, you may add a final update to the comments section of the bug right before you tag it as resolved. Include a description of what exactly was wrong with the bug, and what steps were taken to fix it. This data can be extremely helpful if the solution for said bug was incorrect, or created a bug in a different part of the engine.

Debugging

There are a collection of debug features built into the engine. The developers use these tools extensively as well as the tools provided by Visual Studio 2008 to debug errors in the code base.

Development Console

The in-game Development Console provides a way to manipulate game objects in real time while the game is running as well as run any type of logic that needs to be debug. It is also useful for performing stress tests and running tasks that need to be debugged at certain times, such as connecting to another player, for example.

Debug output

There is a standard console window which provides debug output. This output can accessed within the engine by using any standard method of printing, as well as a macro, ConsoleMessage, which prints to the debug output as well as the Development Console. ConsoleMessage is useful because it can be easily compiled out of the engine. The Memory Manager system (see Memory Management) and the Object Manager (see Object Management) use this debug output extensively.

Debug Drawing

The graphics engine provides an interface for debug drawing wire frame Cubes, Spheres, Planes, Lines, and Points. The Physics engine uses this interface extensively to debug draw physics objects and their intersections. Debug drawing can be enabled and disabled in real time.

AntTweakBar

AntTweakBar, in addition to being use as our level editor interface, is used to display real-time debug information. The information that it displays is completely configurable. Currently it displays frame rate, frame time, and percentage consumptions of frame time by the major systems in the engine.

Unit Test Framework

There is a collection of useful macros in the engine that can be used to unit test modules of newly written code. Unit tests written with these macros are optimized out in release mode, and are also completely agnostic from the entirety of the engine.

Tools

The developers are using the following tools to perform certain tasks.

AntTweakBar

The AntTweakBar library is being used to display real-time debug information, as well as an interface for the Level Editor.

Level Editor

The in-game level editor, in combination with AntTweakBar is used to build new archetypes, arrange them in the game world, place lights, and to save and load levels for modification or creation.

Model Viewer

The model viewer is a crucial part of the art pipeline that allows artists to view how their 3D models will look in the engine, as well as automatically export the .FBX files to the custom binary .mesh file for loading into the engine.

Team Website

The entire Derelict team has an extremely efficient information pipeline in the form of a website using Google Sites. The website is used for many different things including:

- Hosting important and relevant files
- Emailing the team using Google Groups
- Submitting tickets
- Checking schedules
- Reading help articles
- Reading the Game Design Document
- Check recent SVN history
- Checking team members' contact information
- Checking what team members are currently working on
- Logging and tracking meeting notes and tasks
- Providing quick access to art assets for critique, etc.

Custom Bug Tracking System

See Bug Submission Guidelines.

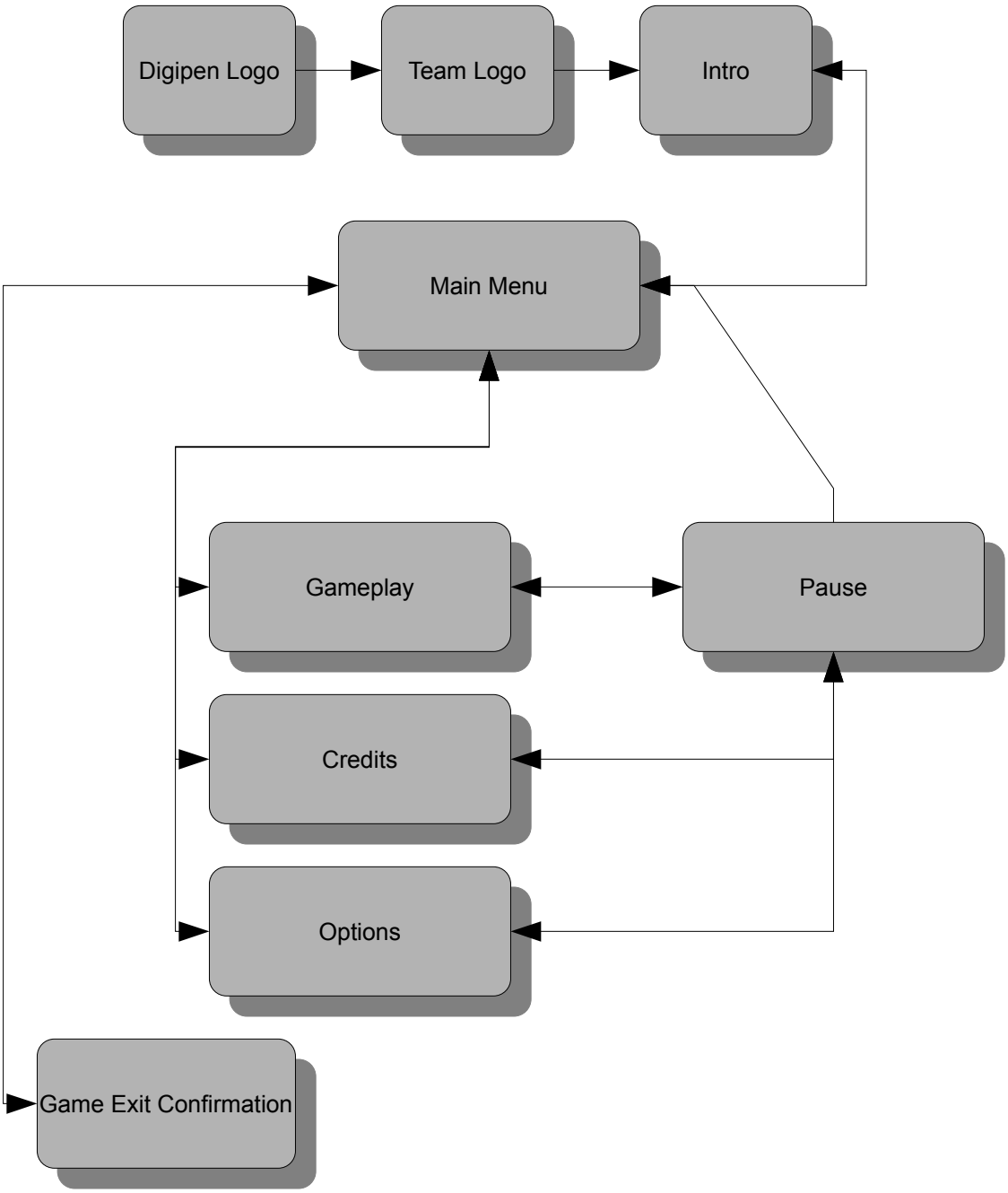
Heart Rate Monitor

The engine uses the Biometric System (see Biometric) to interface with a custom built heart rate monitor which is used to detect the player's heart rate and GSR (Galvanic Skin response).

The current iteration of the prototype is using a LilyPad Arduino Simple Board along with a Bluetooth Mate Silver, and a polymer lithium ion battery to power the unit. Pulse detection is performed using an infrared emitter and detector which reads across a finger. GSR is measured with two copper foil strips attacked to Velcro bands that are meant to be wrapped around two different fingers. The unit contains several resistors, capacitors, and at least one operational amplifier. There are leads which push data into the Arduino and over FTDI/USB or Bluetooth to be sent to the pulling application (in this case, our Engine.) Data packets which contain the biometric information (Y values for the two wave forms) are custom constructed and is performed serially. The wave forms will be processed within the engine. The Arduino is programmed via an FTDI/USB cable.

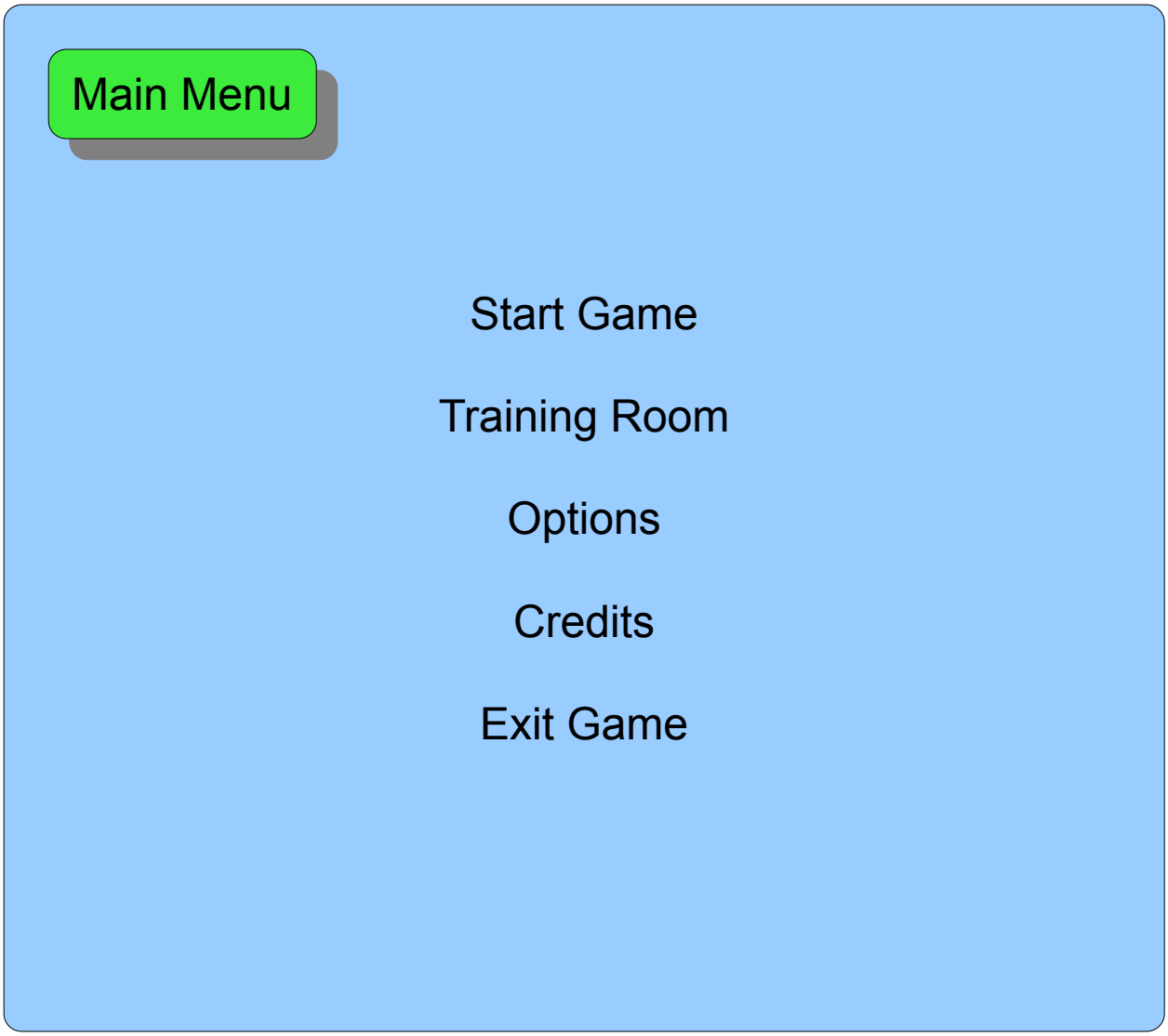
Appendix A: Interface Flow

Flowchart

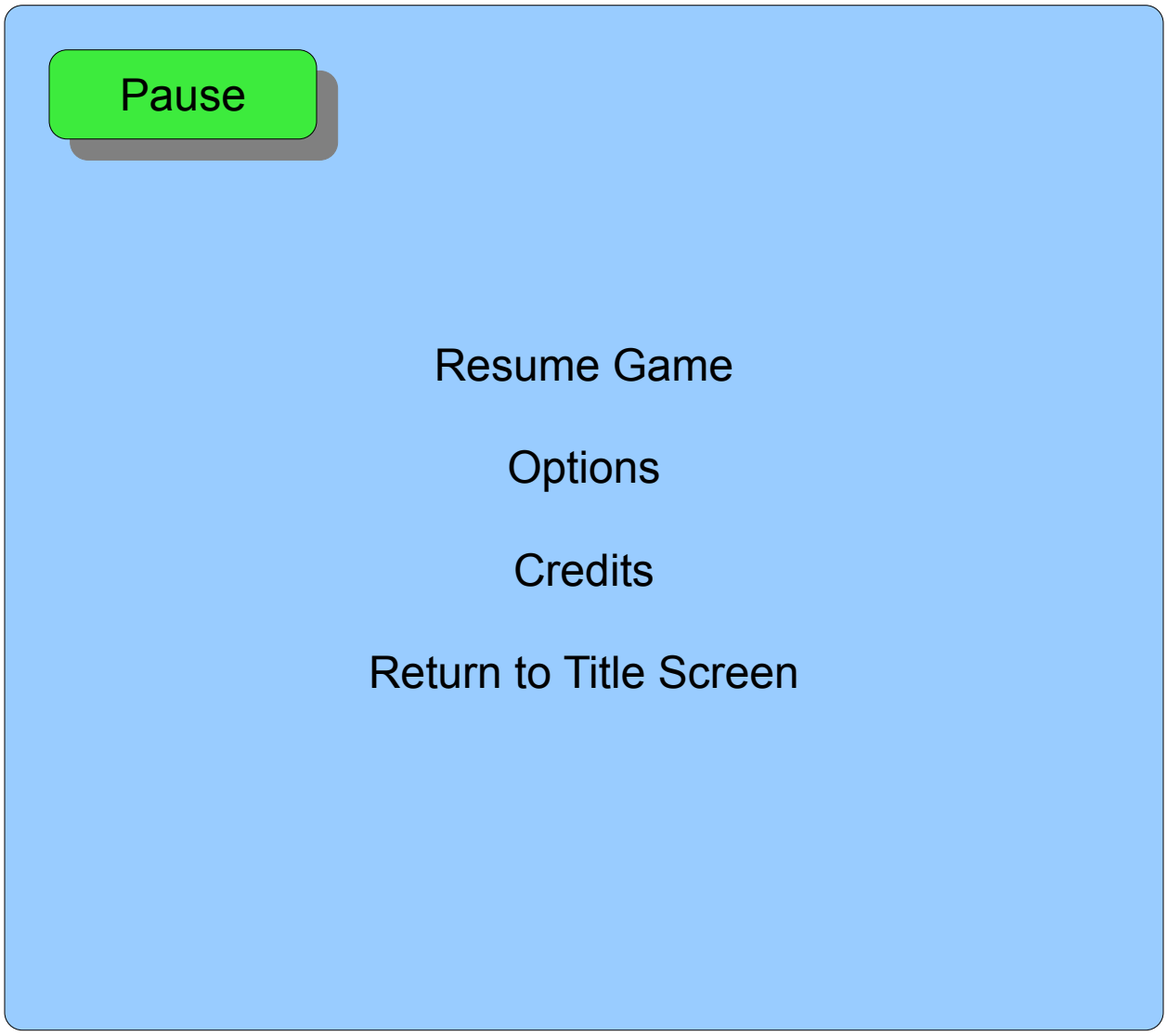


Mockups

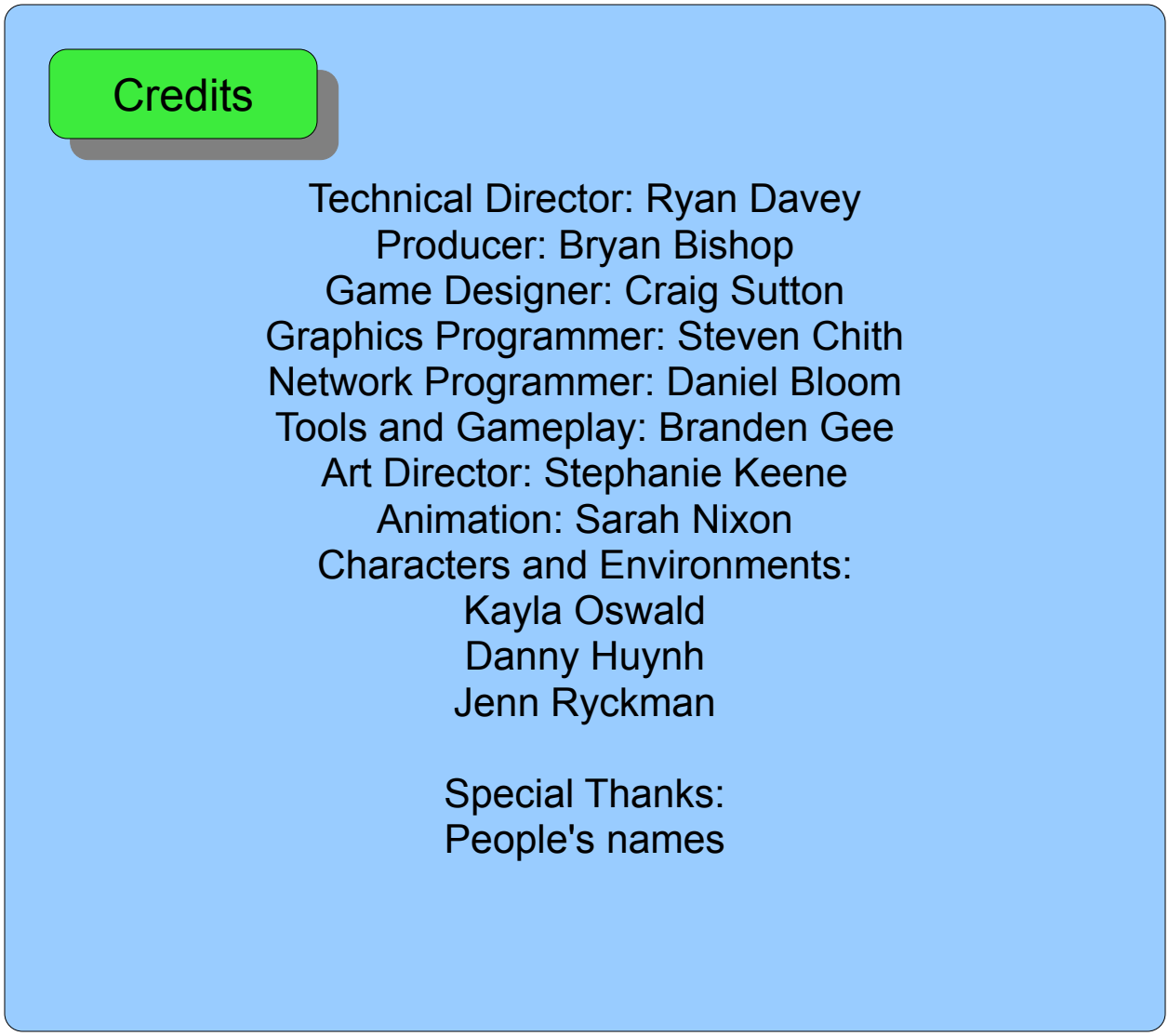
Main Menu



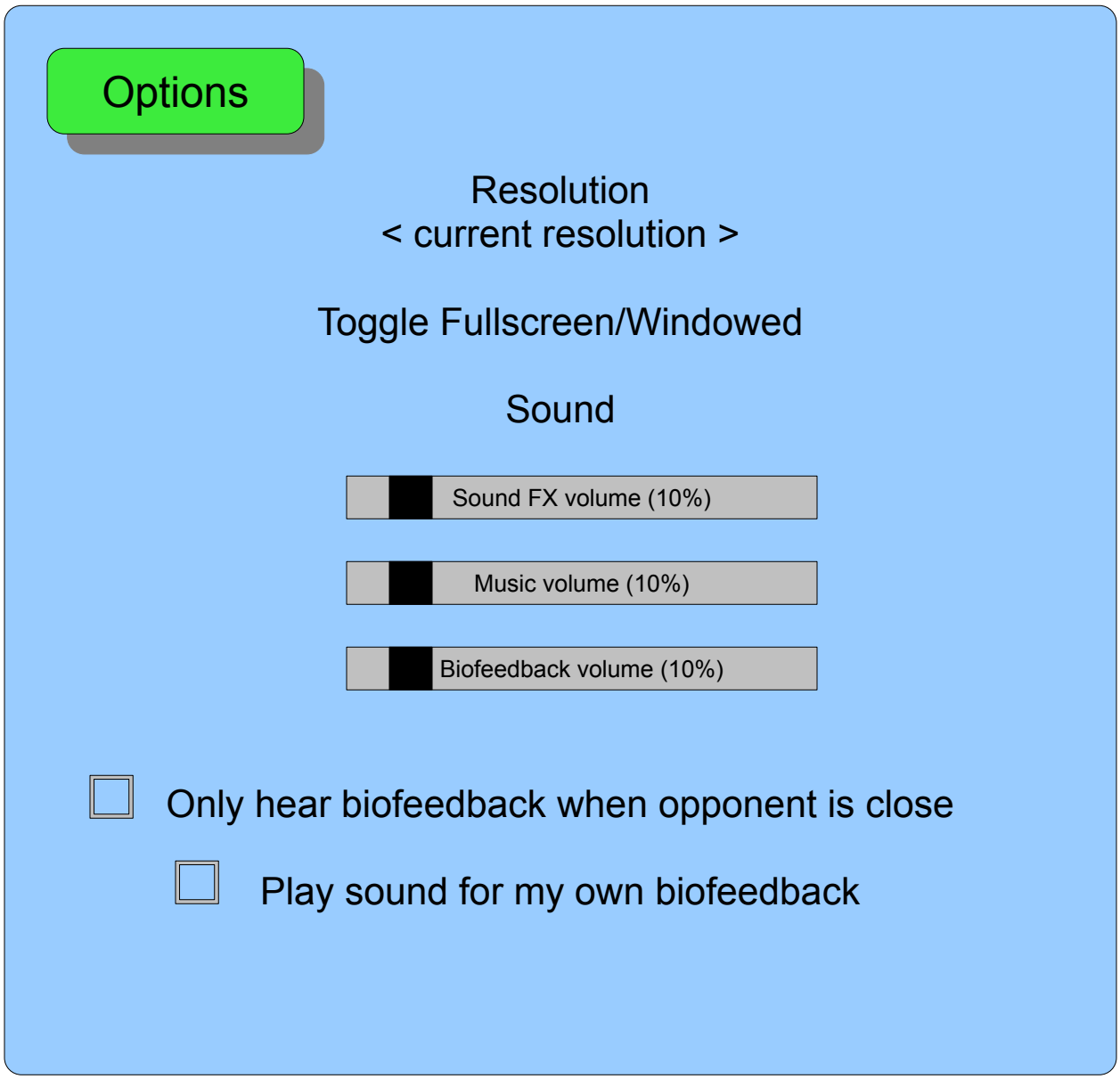
Pause



Credits


Credits
Technical Director: Ryan Davey
Producer: Bryan Bishop
Game Designer: Craig Sutton
Graphics Programmer: Steven Chith
Network Programmer: Daniel Bloom
Tools and Gameplay: Branden Gee
Art Director: Stephanie Keene
Animation: Sarah Nixon
Characters and Environments:
Kayla Oswald
Danny Huynh
Jenn Ryckman
Special Thanks:
People's names

Options


Options
Resolution
< current resolution >
Toggle Fullscreen/Windowed
Sound
Sound FX volume (10%)
Music volume (10%)
Biofeedback volume (10%)
Only hear biofeedback when opponent is close
Play sound for my own biofeedback

Game Exit Confirmation