Conglopoly 2

GAM 400

Fall 2012

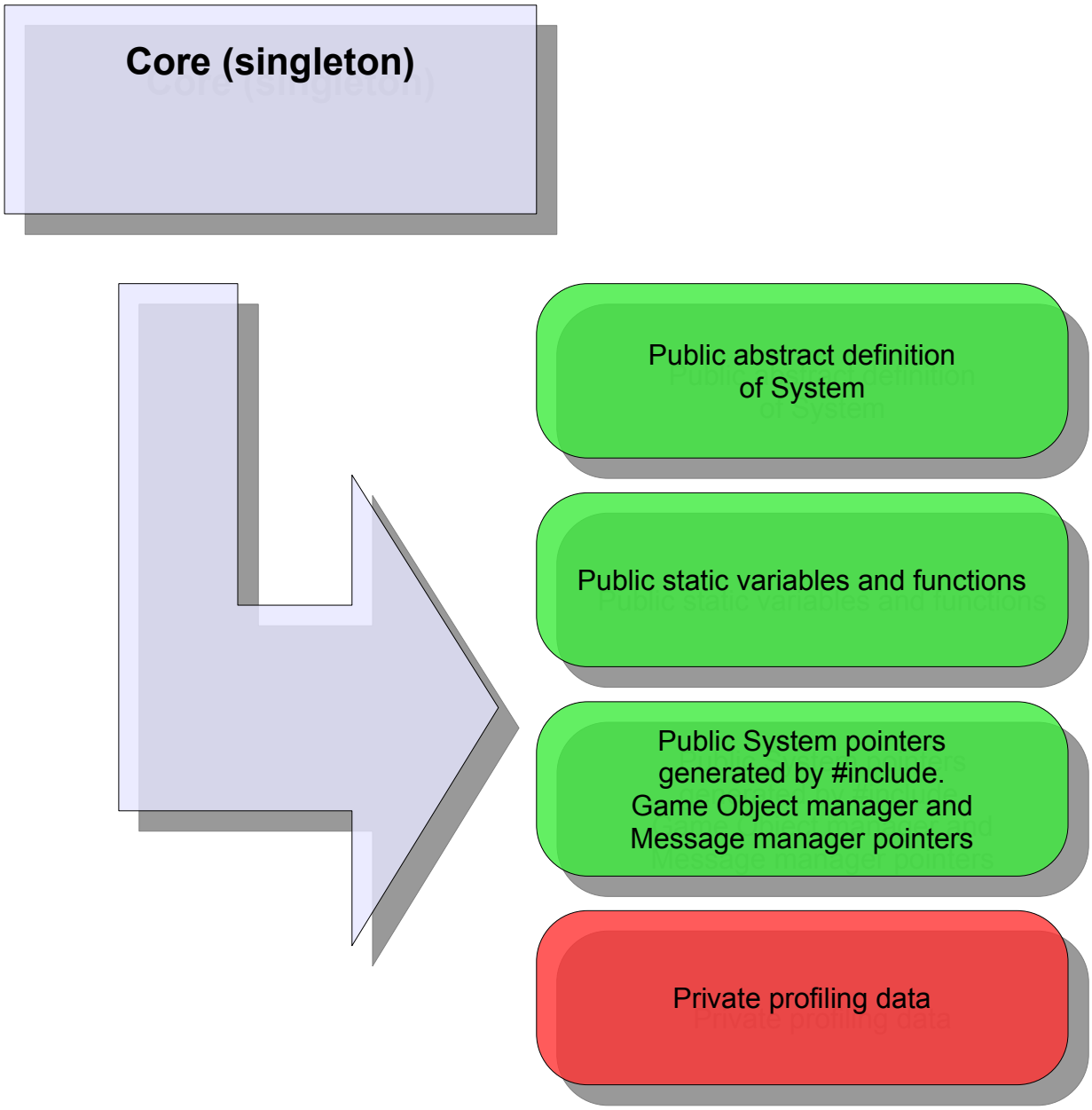
Ryan Davey – Technical Director
Robert Srader – Tools Programmer/Producer

Table of Contents

Overview

The Engine is built from the ground up using only cross-platform APIs for Graphics, Audio, and other systems.

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, game object manager, and message manager pointers:** Pointers to systems may be registered to be added in the core by adding a register call to SystemNames.h. This file is used to generate the proper system pointers using the #include pre-processor directive. The core also contains pointers to a message manager for managing messaging tasks and a game object manager for generating and destroying game objects.
- **Private profiling data:** This data is also generated by the #include pre-processor directive and SystemNames.h. If profiling in the core is enabled, this data is populated and can be displayed in a custom AntTweakBar.

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 the SFML v1.6 library's audio module 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.

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 SFML v1.6 and GLEW 7.0. SDL is being used which combined, provides texture loading and processing functionality to the Engine. The graphics system will also use SFML v1.6 for text rendering. For more information about the Graphics system, see Graphics Implementation.

Input

Input utilizes SFML v1.6 to intercept input data from the operating system which is cross-platform. Once intercepted, the events 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 is not specifically a system, but a set of implementations that are 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, multimap, 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 (print to console) of the state of the object allocators.

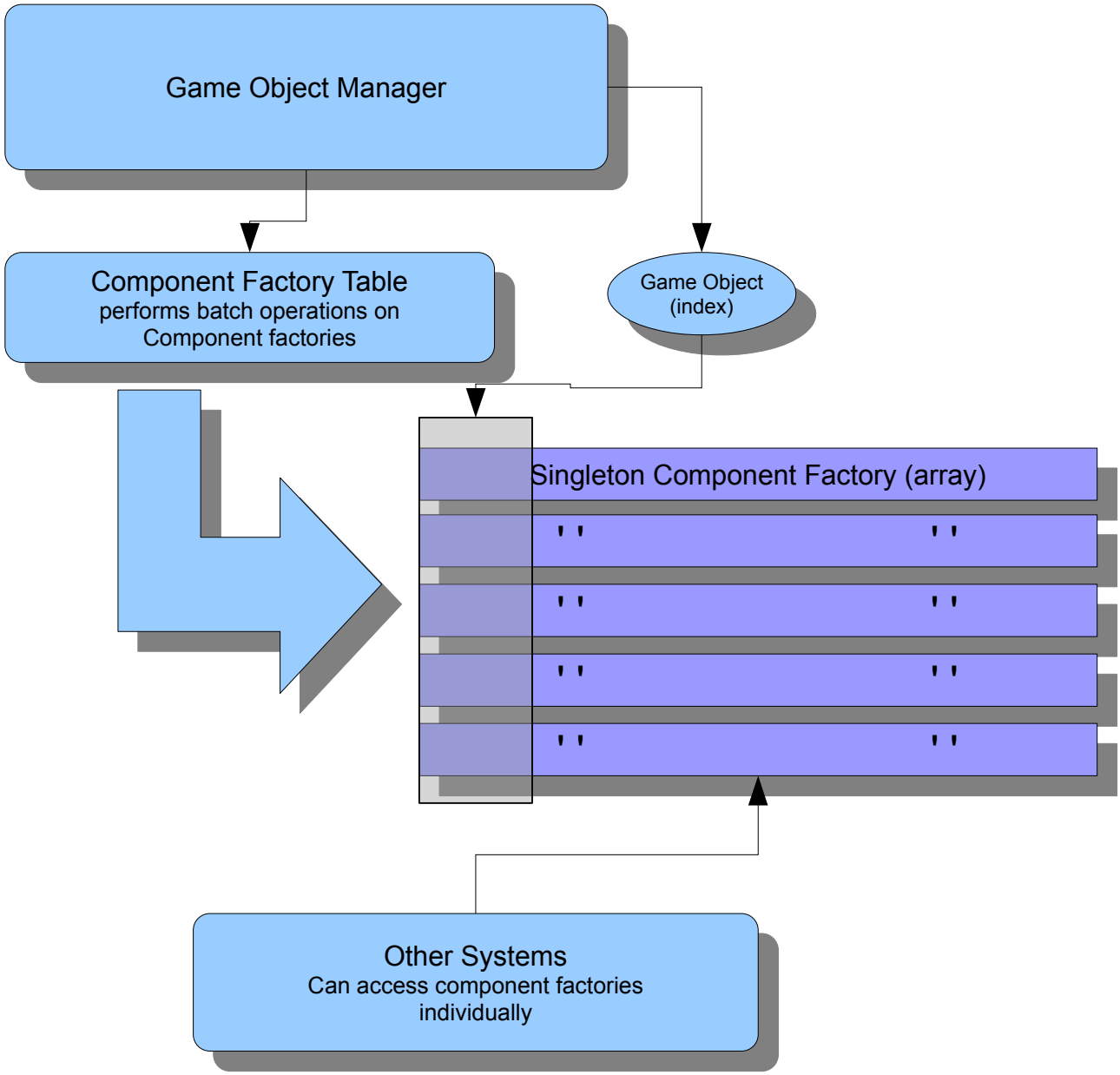
Messaging

The messaging system is not specifically a system, but a separate manager that is accessible through the core. It maintains a table of the objects in the engine that are registered as a message listener. The table is organized by a filtering system. When a message is broadcast, the system calls the message handling functions of each necessary listener. The messages 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, so the other modules in the engine can cache and manage messages however they like. This is particularly important for the

UI System

The user interface system base code is completely custom in C++. It provides a robust set of features for a 2d screen-space UI that will be rendered as the final pass of the deferred shading pipeline. The UI system supports functionality for Text entry fields, text boxes, buttons (with custom textures), and pop-out menus. The system is fully manipulable through C++ interfaces, as well as with Javascript.

Object Management



The object manager is not specifically a system, but a separate manager located in the Core. It 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.

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 three different geometric primitives. Axis Aligned box, Oriented box, and Plane (half-space) since it is a voxel based engine. Broadphase is implemented using an octree. See Physics Implementation.

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, subsequently printing errors and warning messages generated by Javascript. The development console also has functionality to register custom callbacks in C++.

Graphics Implementation

The Graphics uses OpenGL for rendering dynamically lit voxels. Rendering is performed with a deferred shading method and the phong lighting model with cell shaded effect and edge highlighting post-processing effect.

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 emissives.

Asset Management

Since the graphics are voxels, most assets are generated in a programmatic fashion, however, “models” can be saved and loaded from binary files which contain a list of voxels with position and scale information.

Behavior Implementation

There will not be many complex behaviors. The main behavior that is needed is a state machine to control the NPCs.

Customer Behavior

With the current game concept, the customers will be NPCs that roam a store and decide which products they want to buy, if any. This behavior will be mostly programmed in Javascript, due to its flexible nature (making state machines more robust.)

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.

Collision Detecting and Resolution

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

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

Rays provide a way to perform ray-casting for shooting logic, and provides intersection points with physics bodies.

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.

Broadphase

Voxels and bounding boxes will be placed into an octree to perform broad-phase collision detection

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.

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.

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 typedefs defined in Precompiled.h. This is for cross-platform purposes.

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, Planes, Lines (Rays), 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, is used to display real-time debug information. The information that it displays is completely configurable.

Tools

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

AntTweakBar

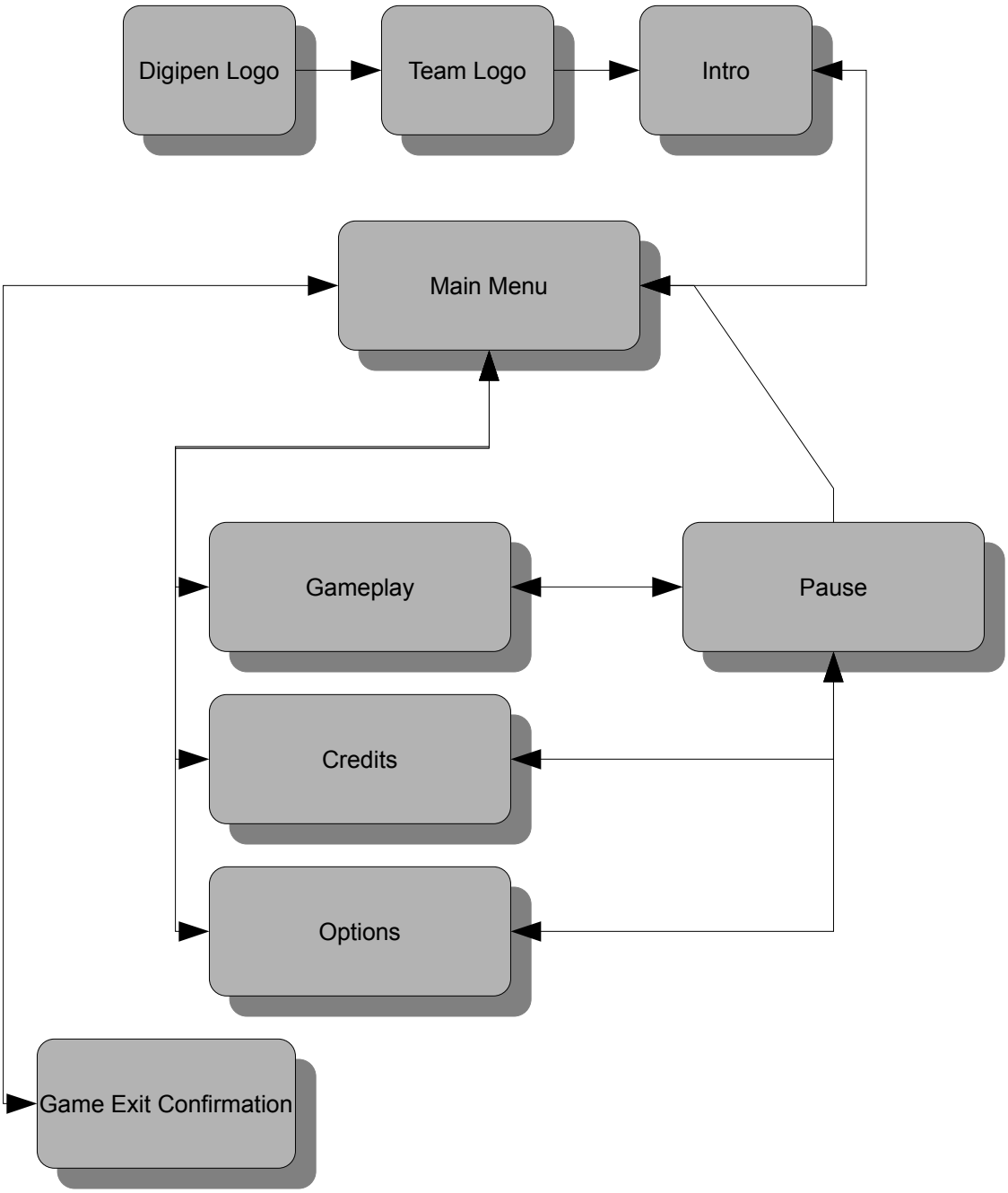
The AntTweakBar library is being used to display real-time debug information, as well as an interface for the Level Editor.

Model Editor

A voxel model editor that is activated in-engine. Uses AntTweakBar or custom UI system to provide tools for making voxel models. User will be able to put together voxels, change color and size of the voxels, and save/load other voxel models.

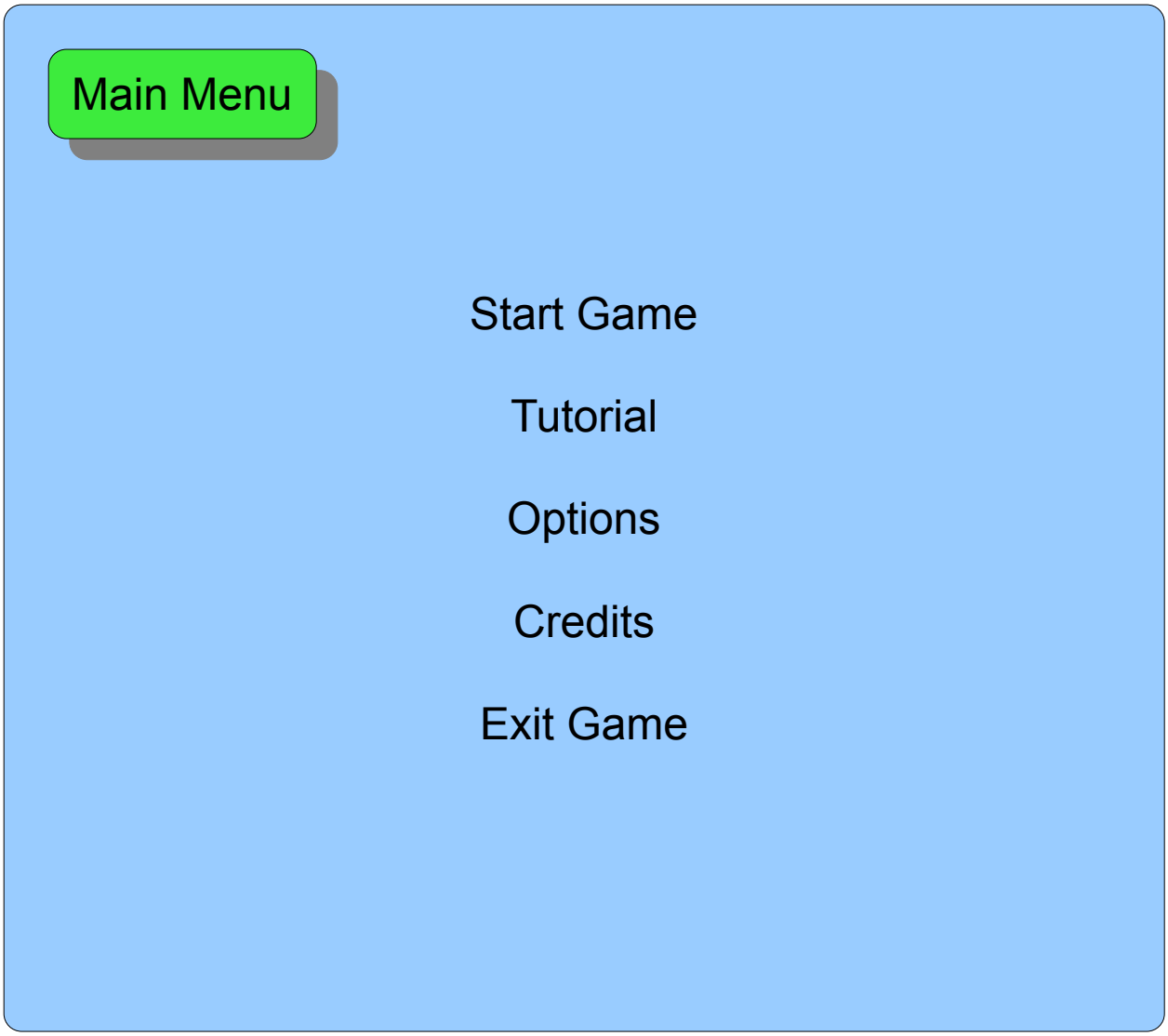
Appendix A: Interface Flow

Flowchart

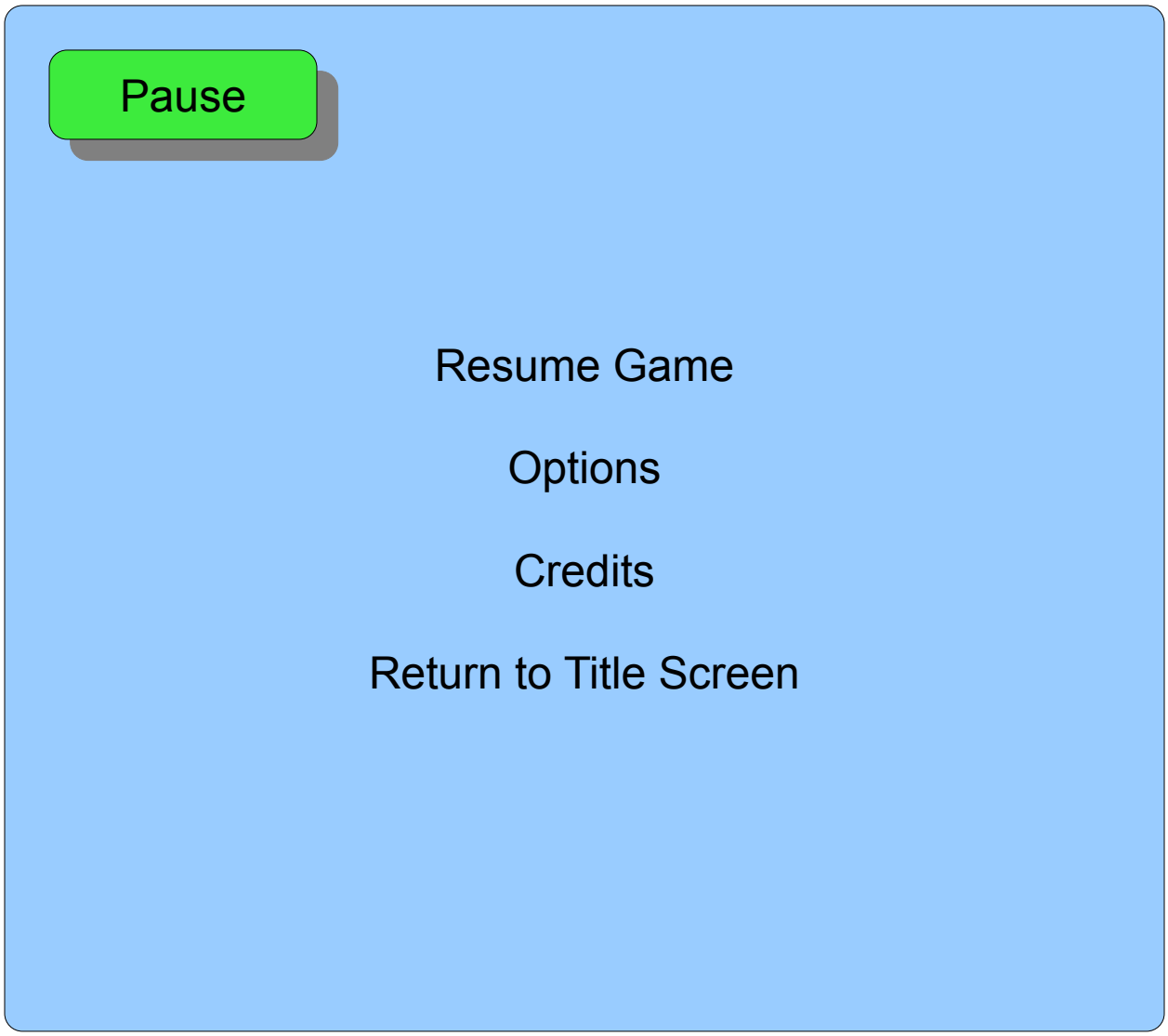


Mockups

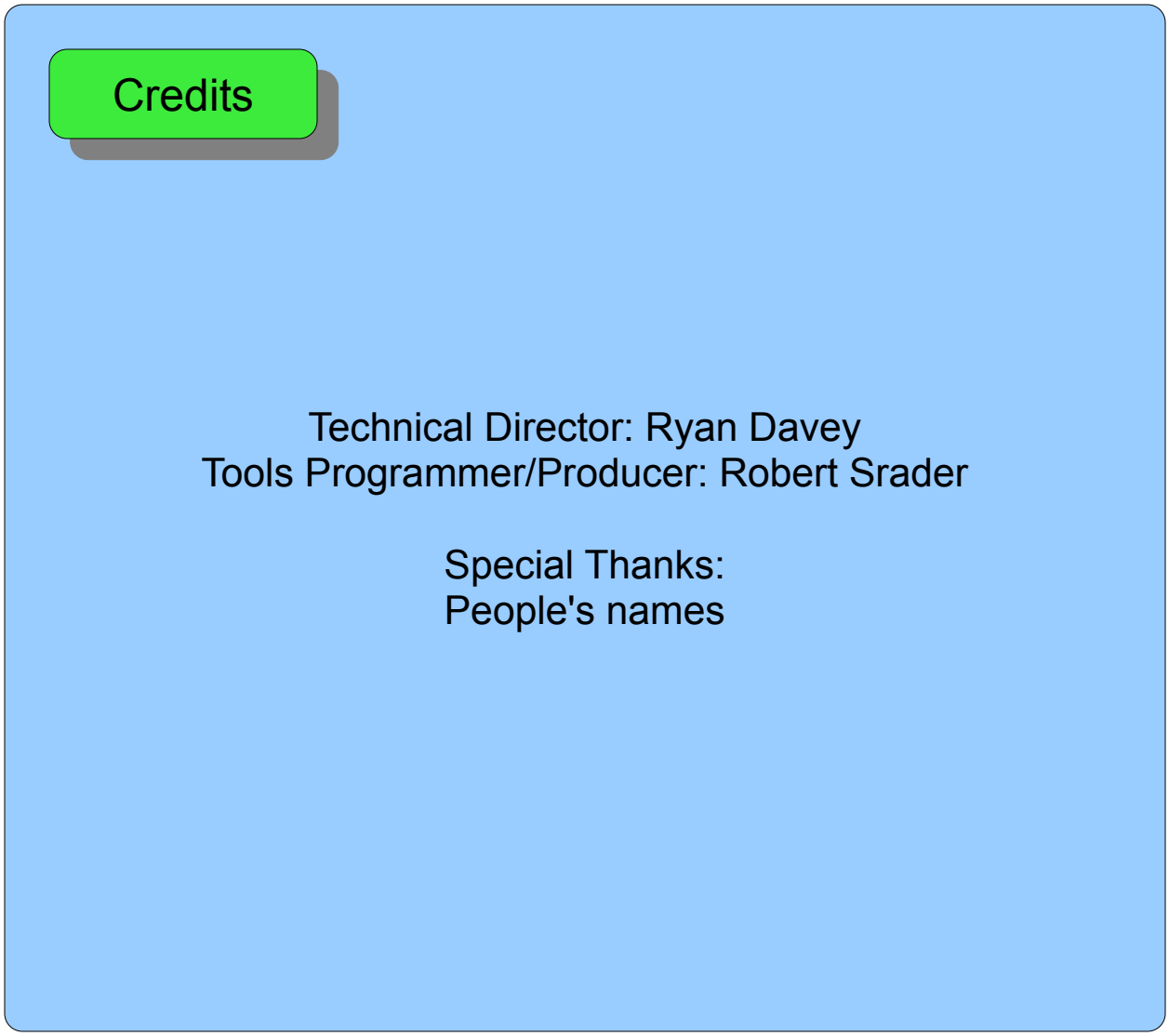
Main Menu



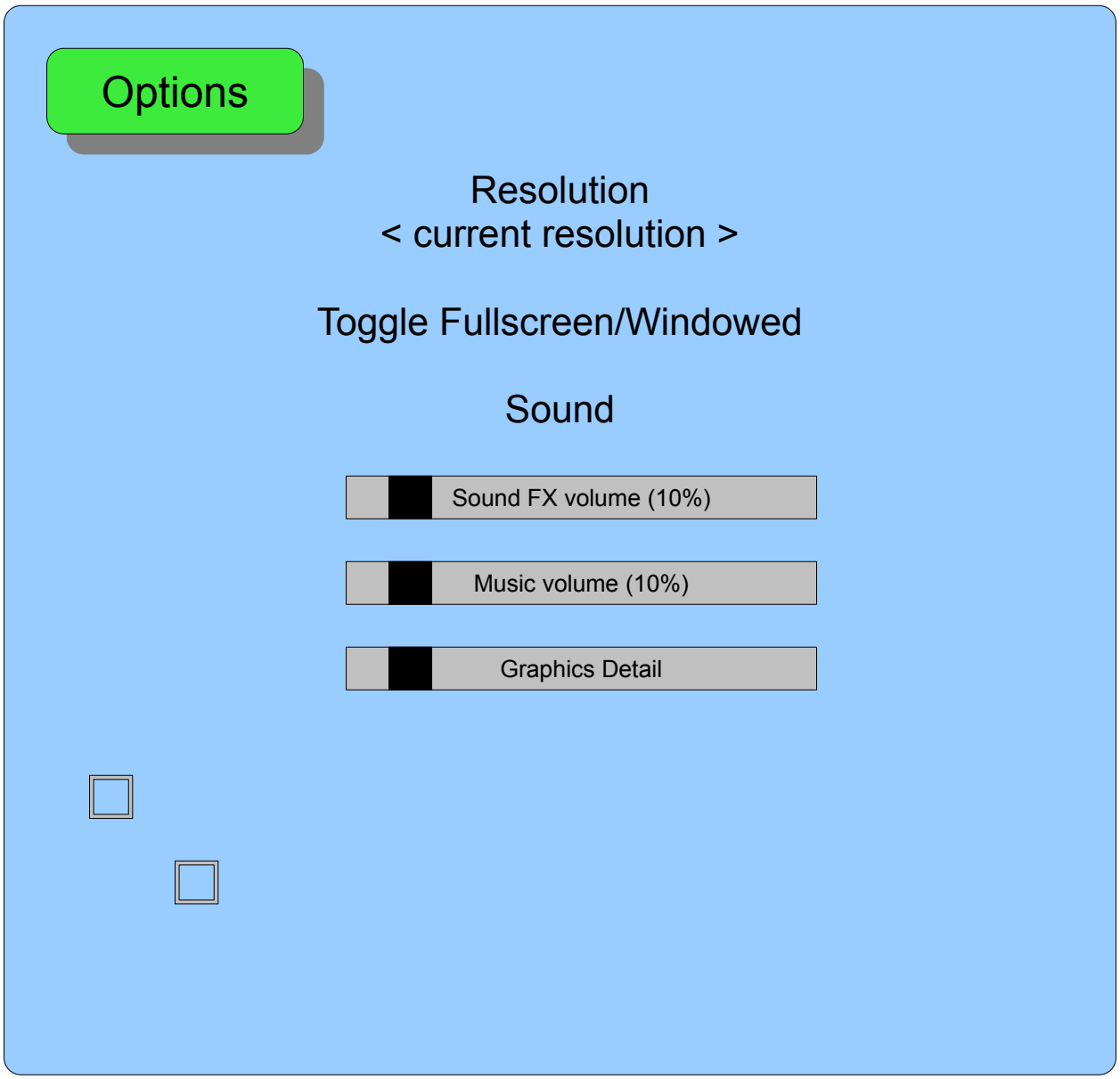
Pause



Credits


Credits
Technical Director: Ryan Davey
Tools Programmer/Producer: Robert Srader
Special Thanks:
People's names

Options


Options
Resolution
< current resolution >
Toggle Fullscreen/Windowed
Sound
Sound FX volume (10%)
Music volume (10%)
Graphics Detail

Game Exit Confirmation