# CRASH
# A real-time 3D game engine designed to make game development easy

*By: Christopher Patton*

*6/16/2015*

## Abstract

*Developing games on top of commercial game engines is difficult because the projects are too large to quickly understand. We present* ***Crash****: an alternative solution to game development that empowers developers to begin working on a game with very little introduction to the project by building a small, extensible, and modular game engine. Design patterns such as dependency injection and interface-based development encourage simple, understandable code and empower developers to divert their efforts toward the playable components of their games.*

# Introduction

Video game development is a difficult and rewarding field. The medium presented by video games give game designers the opportunity to immerse players in a set of ideas in ways that no other technology allows. Players have the ability to actively shape the environment, while game designers have the power to guide their players through the world that they have constructed.

Real-time, three-dimensional video games are most commonly created using game engines, which *"abstract the [...] details of doing common game-related tasks, like rendering, physics, and input."* [4] Using a game engine allows developers to spend less time writing and maintaining these routines so that they *"can focus on the details that make their games unique."* [4] Therefore game engines, by efficiently handling their responsibilities, empower game developers to explore the medium of video games with greater depth than they otherwise would have been capable of.

Game engines assume a wide variety of responsibilities, which often causes them to grow into bloated, monolithic applications. Developing games using traditional game engines is a daunting task because they have such a high learning curve. [5] The desire to shallow this learning curve was the motivation behind the development of *Crash.*

*Crash* is a three-dimensional real-time game engine built with intention of offering developers a simpler and more approachable interface than other game engines. *Crash* is broken up into several modular libraries that allow users to take advantage of any subset of the provided features without forcing them into a specific development ecosystem. This allows users to define their own learning curves by restricting the amount of code that they need to interface with. *Crash* provides window management, user input events, textured mesh rendering, collision detection, and game loop management.

## Related Work

Game engines are the Swiss Army knife of real-time application development. They are commonly used for real-time rendering, physics simulation, scientific research [1], and, of course, video games. In order to be useful in so many types of applications, game engines have to provide developers with an appropriate feature-set backed by adequate abstraction and customizability.

Unfortunately, abstractions and large feature-sets usually incur a cost. [6,7] As a byproduct of having large feature sets, game engines tend to have bloated interfaces. The scale of such projects becomes unapproachable for new developers, which introduces a huge barrier to entry for anyone trying to use a new game engine toolset. [5]

The open-source projects *OGRE* [2] and *OpenSceneGraph* [3] are very mature, feature-rich engines that provide developers with this level of abstraction that makes them overwhelming. Both are perfectly capable of producing high-quality end-products, but only after a developer has spent hours poring over documentation, examples, and implementation code.

| Project | Implementation (lines of code) | Documentation (words) | Included technology samples | Classification |
|---|---|---|---|---|
| OGRE | 240,000 | 150,000 | 53 | Small |
| OpenSceneGraph | 620,000 | 11,000 | 175 | Large |

The characteristics of each of these projects are very stereotypical for open-source game engines. The implementations are too large to be fully understood; the documentation is either overwhelmingly large or too sparse to fully cover the features that would be needed for each developer’s use case.

## Solution

*Crash* was designed from the top down to be a modular and understandable framework to take the place of fully-featured game engines for smaller projects. The purpose of *Crash* is to provide developers with a discrete set of libraries that they can use to solve problems commonly handled by game engines.

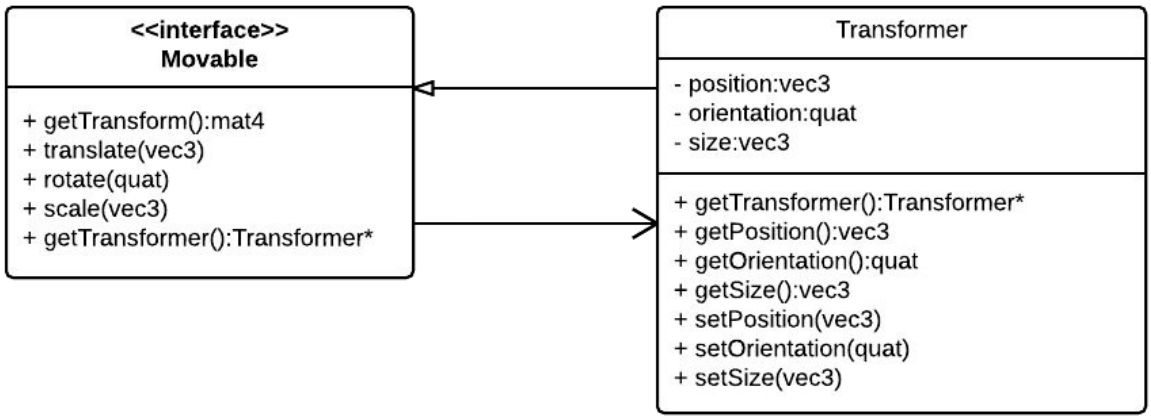
*Crash* is divided into five libraries:

- **common:** a set of utilities used in all of the other libraries;
- **window:** an object-oriented abstraction over GLFW user input events and windowing system (which, itself provides an abstraction over OpenGL contexts);
- **space:** a small collection of classes used in making spatial queries for collision detection;
- **render:** an object-oriented abstraction over OpenGL resource handles and AssImp scene components; and
- **engine:** a generic game-loop manager.

By limiting the scope of the project, utilizing a small set of dependencies, and taking advantage of maintainable development patterns such as dependency injection and reusable interfaces, *Crash* provides real-time application developers with the tools they need while maintaining a small footprint and low learning curve.

## Common

The common library contains a series of useful mathematical tools and constants, the *Plane* geometry representation, the *Movable* interface, and the *Transformer* utility class. *Transformer* is a generic implementation of *Movable* that contains position, rotation, and scale data.The relationship between *Movable* and *Transformer* is bidirectional, as *Movable* requires the composition of a *Transformer*.




Several classes in other libraries implement the *Movable* interface by delegating method calls to a *Transformer* member variable. This design pattern allows a multitude of derived classes to be used as *Movable*'s while segregating the details of the *Movable* implementation to *Transformer*, which divides code complexity into more easily digestible components without incurring any unreasonable performance penalties.

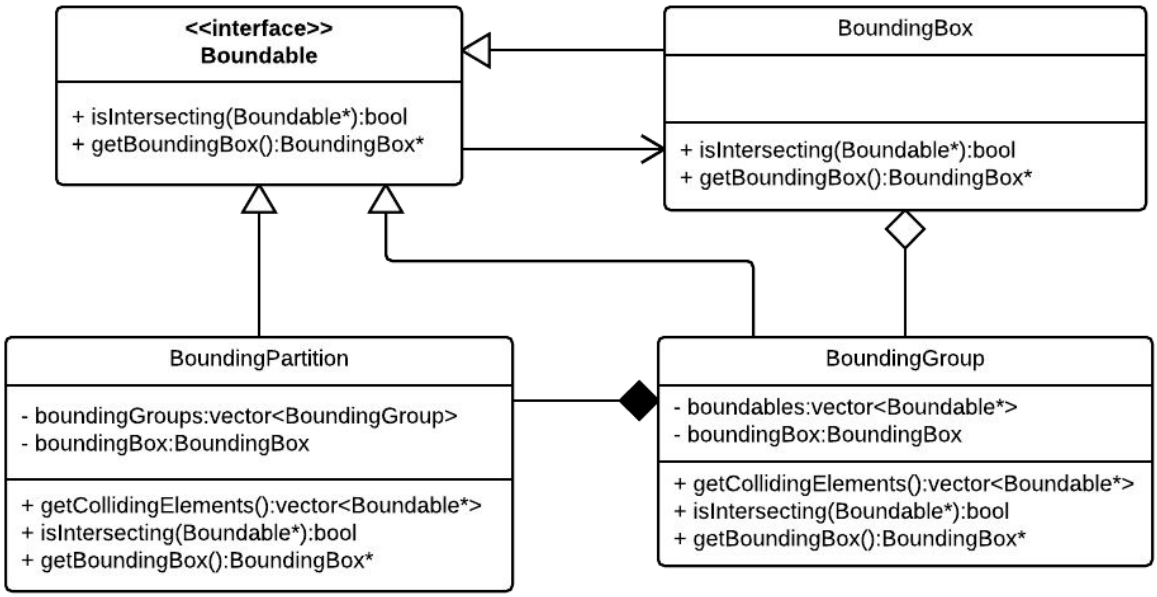
## Space

Spatial queries are an unavoidable component of any physics or graphics application. Spatial data structures such as octrees and bounding volume hierarchies are used in non-real-time applications due to the level of granularity they can be reduced to. However, these structures have a high performance cost in real-time applications, as the hierarchy has to be rebuilt for every frame. Instead, uniform spatial subdivisions (a three-dimensional grid) are more commonly used.

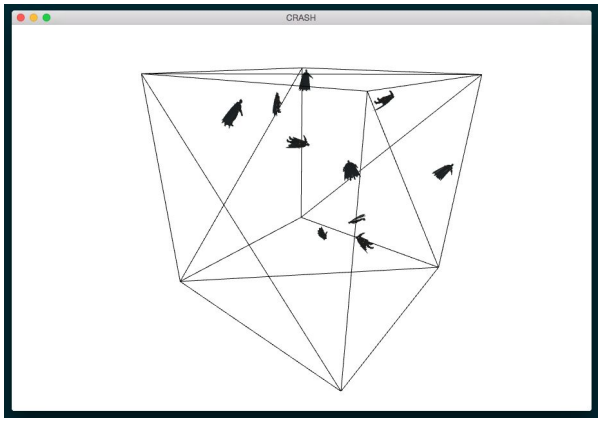
*Crash* uses uniform spatial subdivisions for all of its internal spatial queries. The space library contains three separate classes and one interface to facilitate this model. The top-level abstraction of the spatial grid is the *BoundingPartition*, which dynamically manages composition of a three-dimensional, partitioned collection of *BoundingGroup*'s. *BoundingGroup*'s aggregate a set of objects implementing the *Boundable* interface, which declares a few basic spatial queries which are used by *BoundingGroup* and *BoundingPartition* (both of which implement *Boundable*).

*BoundingBox* is a generic implementation of *Boundable*. The relationship between *Boundable* and *BoundingBox* is bidirectional, as *Boundable* requires the composition of a *BoundingBox*. This allows collisions between any and all *Boundable*'s to be evaluated as collisions between two oriented bounding boxes.

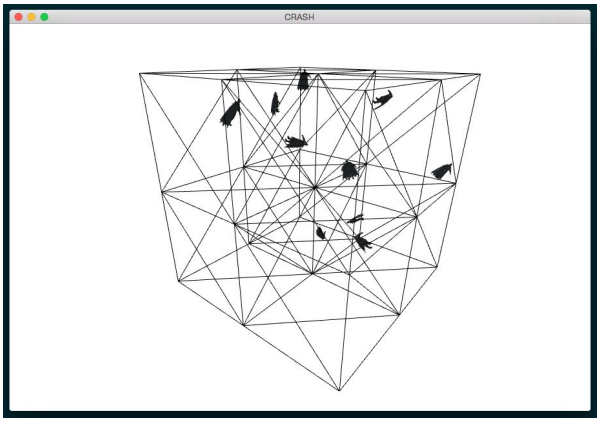
Interface with the spatial query system is achieved exclusively through *BoundingPartition*. *BoundingPartition*'s maintain a structured, partitioned collection of *BoundingGroup*'s of uniform size, position, and orientation. Each of these *BoundingGroup*'s is independently queried when searching for *Boundable*'s that satisfy the current query, and their result sets are aggregated in a single list. In the event that a *Boundable* contained within the *BoundingPartition* does not reside in any of the *BoundingGroup*'s, the *BoundingPartition* must query these *Boundable*'s directly or ignore them for the given query.



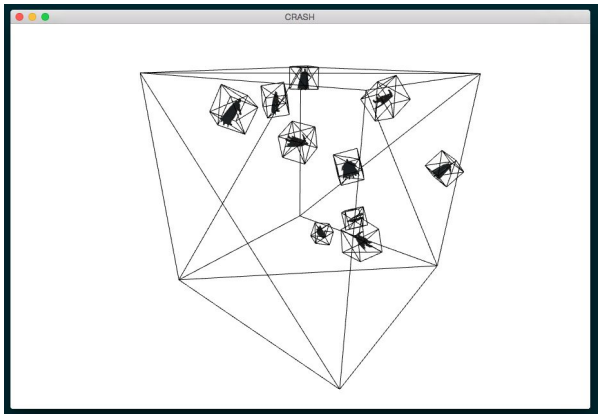

The benefit of implementing the core functionality of spatial queries in *BoundingBox* is that implementing *Boundable* is trivial. This makes it very easy for developers to make their own derivative classes that can be used in spatial queries. It is also very easy to use any or all of *BoundingBox*, *BoundingGroup*, and *BoundingPartition* to create a custom spatial hierarchy. A project that requires non-uniform spatial query system can still use the robust feature-set of the space library with a small amount of custom development work.



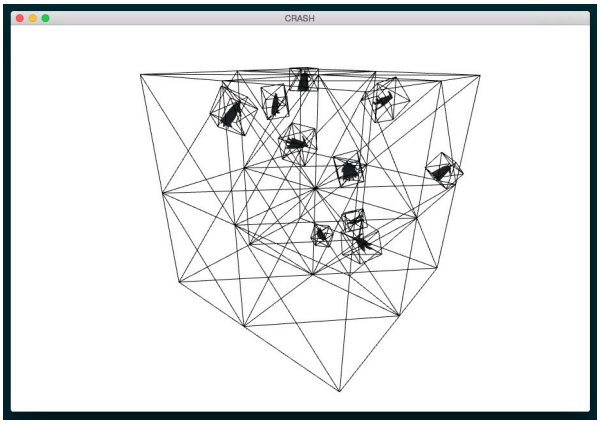

*Rendering a **BoundingPartition** populated with a random collection actors.*



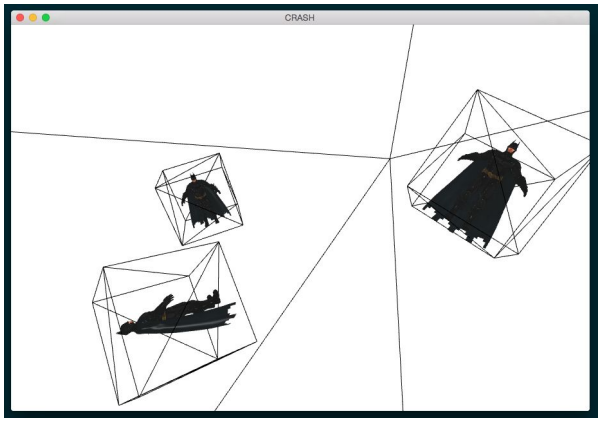

*Rendering all **BoundingGroup**'s in the **BoundingPartition**.*



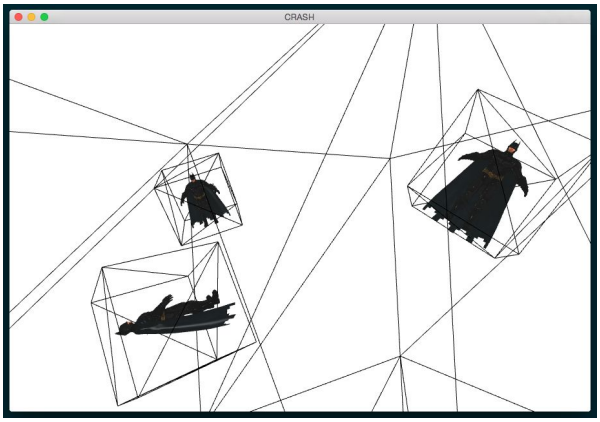

*Rendering **BoundingBox**'s on all the actors in the **BoundingPartition**.*




*Rendering **BoundingBox**'s on all actors and **BoundingGroup**'s.*




*A close-up render of **BoundingBox**'s on actors in the **BoundingPartition**.*




*A close-up render of **BoundingBox**'s and **BoundingGroup**'s.*

## Window

The window library provides a thin object-oriented wrapper around GLFW's procedural windowing system. By nature of including GLFW, the window library also provides an abstraction around OpenGL contexts (as GLFW windows themselves are tied to no more than one OpenGL context each).

Every class provided in the window library extends a base *GlfwAdapter* class, which performs two separate functions: error callback registration and static initialization/teardown of the GLFW runtime. *GlfwAdapter* contains a static list of error callbacks to be called in sequence when an error is occurred. *GlfwAdapter* also contains a static *shared_ptr* to a *GlfwInitializer*, which serves the sole purpose of calling *glfwInitialize* and *glfwTerminate* on construction and destruction.

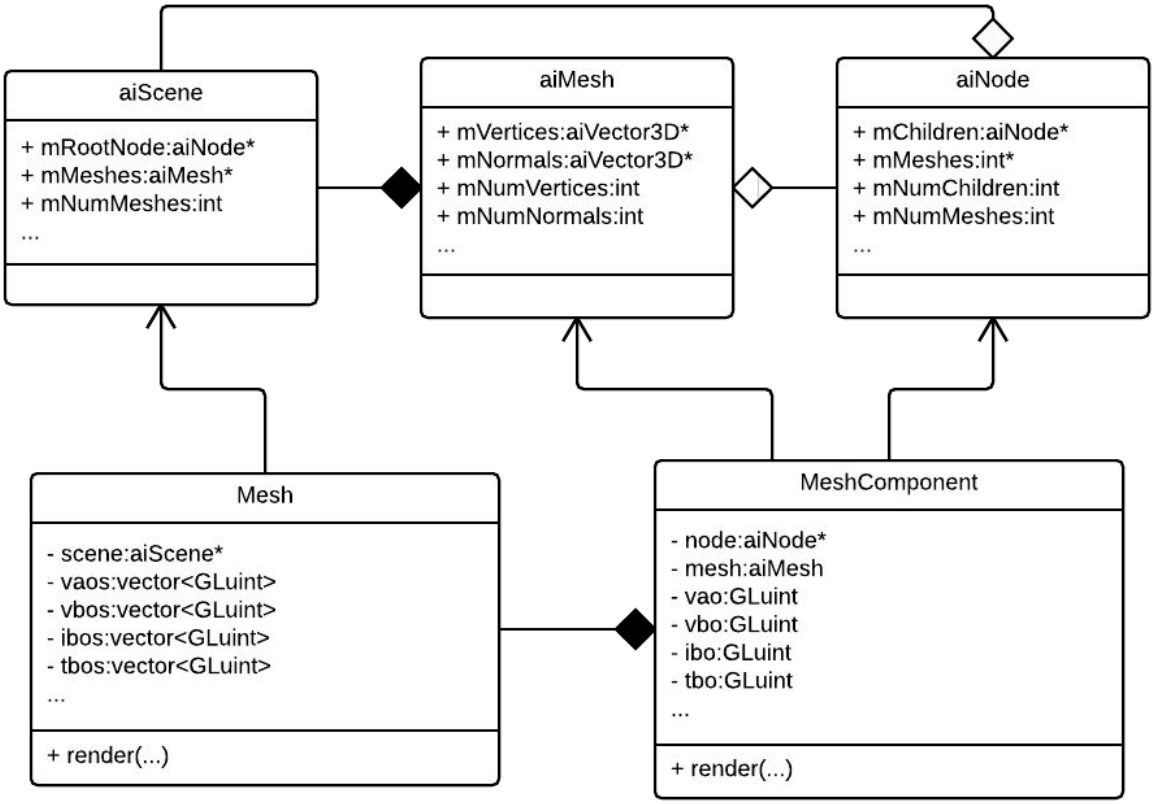
The *Window* class is the only instantiable object in the entire window library. Each *Window* has its own OpenGL context (unless an existing context is specified), and corresponds to a different X11 window. Each window has a series of actions that can be taken on it (minimize, hide, resize, etc), as well as a collection of event callbacks that can be set. These callbacks are triggered when events such as mouse clicks, key presses, window focusing, etc occur. By populating these callbacks and periodically calling *glfwPollEvents* (or *glfwWaitEvents* in an asynchronous application), a developer can achieve event-driven user interaction with very little code.

## Render

Rendering engines can become complicated very easily, especially when they interface with OpenGL contexts. Geometry, textures, shaders, and shader variables all use persistent handles that are global to the current OpenGL context. Creation of this data can be exceedingly complicated, as can the initialization of memory on the graphics device. *Crash*'s render library provides object-oriented wrappers for these data types that handle initialization of the data both on the host device and the graphics device, as well as a simple API to handle complicated procedures like component-based mesh rendering.

*Crash* uses *AssImp* to handle all of its geometry importing. *AssImp* creates a data structure for imported models that contains geometry, material, and textures for each individual component of the mesh, as well as whatever armature structure was defined in the model for skinned mesh animations. These mesh components are related to one another in a hierarchical tree separated by transformation matrices. *Crash* wraps these structures with two classes: *Mesh* and *MeshComponent*.

*Mesh*'s handle the allocation and deallocation of OpenGL handles for all of their composite *MeshComponent*'s. Upon construction, each *MeshComponent* receives one vertex buffer handle, one index buffer handle, and as many texture buffer handles as the underlying *AssImp* mesh needs for its textures. When rendering a *Mesh*, each *MeshComponent* is rendered separately. The *Mesh* calculates the relative transformation matrices needed by each *MeshComponent*, and the *MeshComponent*'s makes the necessary OpenGL library calls to render their associated geometry. Utilization of this composition model prevents users from interacting with the OpenGL context at all, which drastically increases code stability.

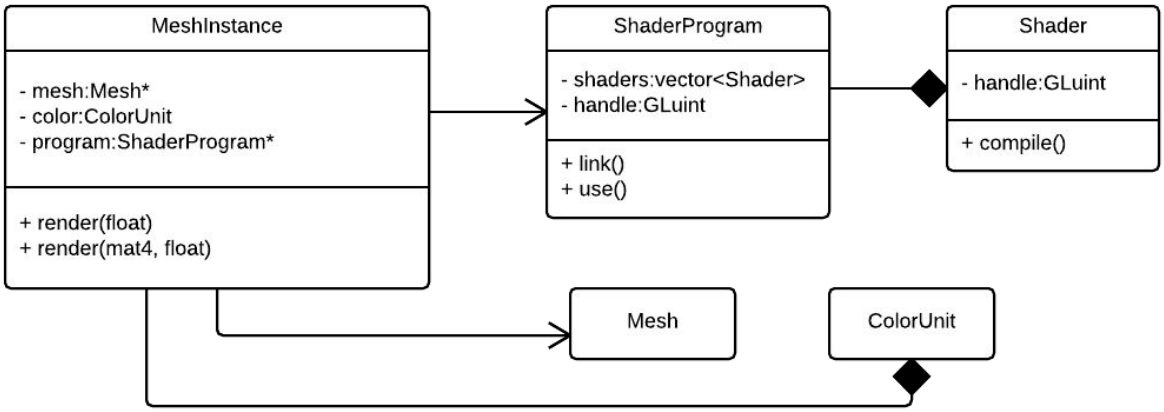



*Crash* handles texturing by using the *STB* library to load texture data referenced in *AssImp*'s data structure for each piece of geometry. Texture loading and decompression is handled during the construction of the *Mesh* object, further reducing the need for users to interact with the OpenGL context.

The render library provides a simple to use mesh-instancing system to allow for easy reuse of geometry and textures. The class *MeshInstance* represents a renderable instance of the *Mesh* specified on construction. *MeshInstance*'s can be rendered differently from their otherwise identical brethren by specifying a different material color or shader program. The *Renderable* interface is defined in this library, which follows the same implementation model as *Movable* and *Boundable*. *Renderable* is implemented by *MeshInstance*, which is the basic renderable type that all *Renderable*'s must contain.

*Crash* encloses shaders and shader programs in object-wrappers as well. Each *Shader* instance compiles its source shader file on construction, immediately reporting errors as exceptions. *ShaderProgram*'s aggregate a set of *Shader*'s and link them against a set of vertex attribute data. Access to uniform variables is controlled by each *ShaderProgram*, and initialization and modification errors are immediately reported as exceptions as well.

*MeshInstance*'s contain member *ShaderProgram*'s and *ColorUnit*'s that they use when rendering. Changing which ShaderProgram is used to render a given *MeshInstance* is as simple as overwriting its member variable. Likewise, the *ColorUnit* can be overwritten in the same way. Because *Renderable* implies composition of a *MeshInstance*, rendering for any object is just as dynamic as for *MeshInstance*.



*A high-poly model rendered using basic functionality from **Crash** and a simple Phong shader.*

## Engine

*Crash* defines a small engine library to unify the components from the other libraries into a cohesive unit. This library contains three classes: *Actor*, *Camera*, and *Driver*. *Actor*'s are top-level implementers of the *Movable*, *Boundable*, and *Renderable* interfaces. *Camera* is a special type of *Actor* that has a series of utility methods for rendering purposes. *Driver* is a game loop controller that aggregates a collection of *Actor*'s and interfaces with them during each update and render cycle according to a dynamic set of user-specified collision, update, and render callbacks.

*Driver* contains a series of objects found throughout the rest of the *Crash* framework. Each driver consists of a *BoundingPartition*, a *Camera*, a *Window*, and a collection of *Actor*'s. Each update cycle, the *BoundingPartition* is queried to see which *Actor*'s are colliding. These colliding *Actor*'s are passed to a series of user-specified collision callbacks. Then, every *Actor* is passed to another series of user-specified update callbacks. Finally, each *Actor* is moved along it's defined (translational, rotational, and scale) velocity for the time elapsed since the last update cycle. Each render cycle, the *Actor*'s in the scene are passed to a series of user-specified render callbacks. Then, each of the visible *Actor*'s are rendered to the *Driver*'s *Window* member.


*A collection of Batmen following scripted paths.*

# Conclusions

*Crash* provides a thorough set of functionality needed for game development. These features free game developers from the need to write their own engine, and instead focus on developing gameplay. *Crash* provides OpenGL context handling, windowing, and user input management through its window library. Performant collision detection can be gained with minimal developer overhead by using the space library. The render library enables painless mesh and texture loading, shader management, mesh instancing, and rendering without the need to interface with the OpenGL context. Customizable game loop management can be gained by using the engine library.

The rendering system in *Crash* proved to be the most difficult component of the engine to develop. There is simply no way to reduce the amount of work needed to render a hierarchical mesh. *AssImp* offers a large amount of convenience by wrapping the complexities of 3D models into scenes, but a very large amount of code is still needed to render a mesh hierarchy. *Crash* does an excellent job of concealing the disaster of OpenGL interaction by providing a simple public interface to mesh creation and instancing. However, the implementation of the components of the rendering system are exceptionally extensive and complex.

The dependency injection and interface development design patterns are major contributors to the quality of *Crash*. By requiring dependent object instances to be specified in object constructors, *Crash* forces developers to understand the relationship between entities at compile time. This helps answer basic questions about resource ownership and library dependencies that would increase the difficulty of development if left unanswered. Providing trivially implementable interfaces makes it easy for developers to build systems that interact cleanly with the features of *Crash*. The details of these specific interfaces encourage a composition-based relationship structure, which is generally considered to be better than inheritance-based models. [8]

# Future Work

There is a wide variety of future development that would benefit Crash. From this set, five primary features seem to be most appropriate for the framework: view frustum culling, skinned mesh animations, a state management system for Driver, multi-skin texture support, and event-based collision detection (instead of polling-based).

View frustum culling is the most immediately useful of these features. The spatial queries provided in *Boundable* could be expanded to contain a visibility query against a viewing frustum. The algorithm used for view frustum culling is very well known. It involves comparing the extreme points (the opposite corners whose diagonals are more orthogonal to the plane) of the oriented bounding box to the planes of the viewing frustum.

Skinned mesh animations are a very important component of any game. Without them, components of the game remain lifeless and non-immersive. Implementation of skinned animations is a very hard problem, with implementation details varying between model formats. Rigged models contain an armature of “bones” defined by the model artist. Model vertices are influenced by the bones of the armature. Through skinned animation, movement of the bones adjusts the position of the vertices, creating the impression of movement.

The state management system for *Driver* is the easiest of these to implement. A *State* interface would be created that listed *onEnter* and *onExit* functions, as well as all of the callback registration currently found in *Driver*. *Driver* would have to be modified to operate a current *State** member, but the overall implementation of *Driver* would become drastically simplified.

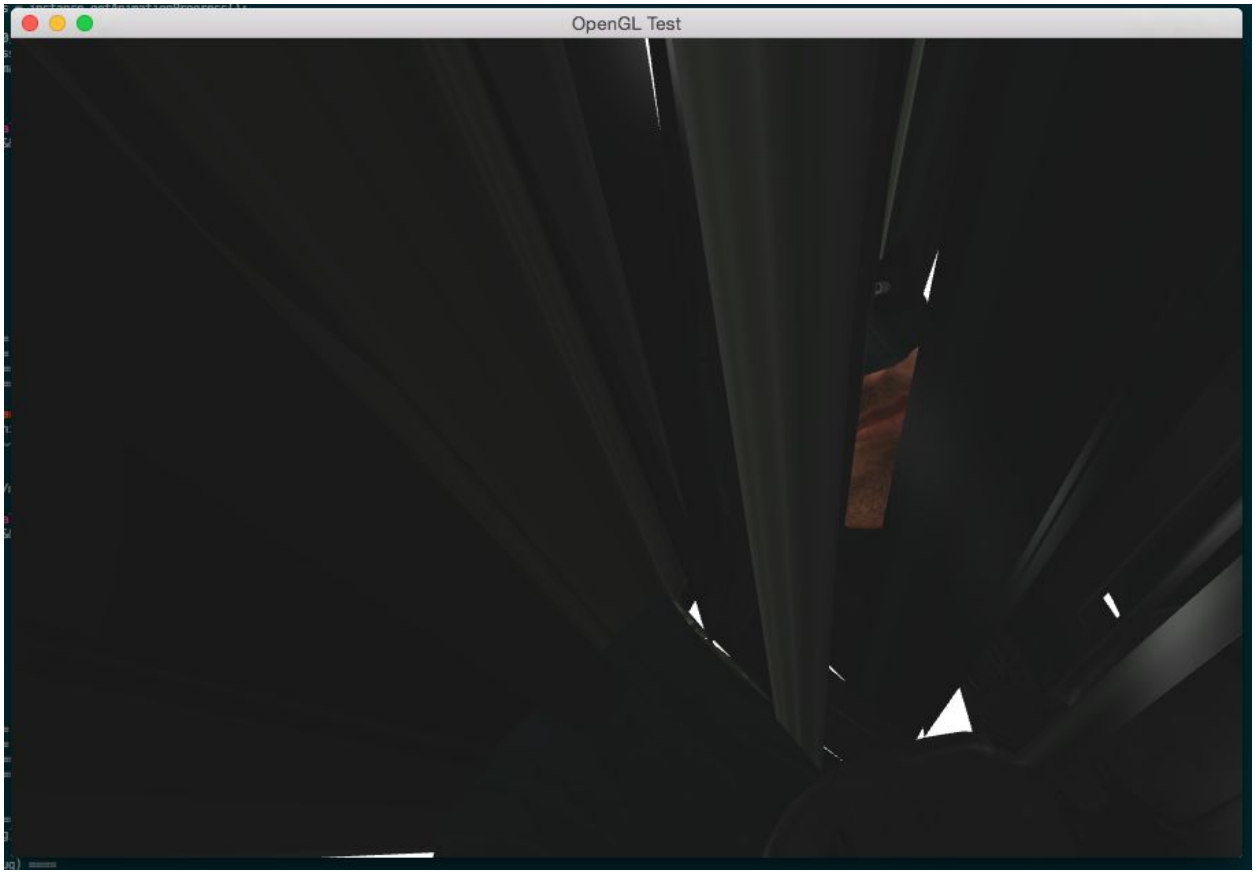
Multi-skin texture support could be accomplished quite easily. AssImp allows several textures to exist for each channel. *Crash* currently uses the first textures found in each of the displacement, normal, ambient, diffuse, specular, and shininess texture channels to populate the graphics device. However, if every texture found in each channel were used, the notion of a skin could be created by mapping each channel to the texture index to use for each *MeshInstance*.

Event-based collision detection would require tracking which *Boundable*’s are colliding at each update cycle and only notifying the user of the library when this state changes. Implementing this in *Driver* would incur a heavy cost for all users, even those that only care to see collision polling reported. A more sustainable solution is to allow users (or *State*’s) to implement the collision tracking to achieve event-based collision detection.

# References


[1]http://publicvr.info/publications/Lewis2002.pdf
[2]http://www.ogre3d.org/
[3]http://www.openscenegraph.org/
[4]http://www.gamecareerguide.com/features/529/what_is_a_game_.php
[5]http://www.learn-cocos2d.com/2012/05/the-game-engine-dating-guide-how-to-find-the-right-engine-for-your-game/
[6]http://programmers.stackexchange.com/a/190832
[7]http://www.joelonsoftware.com/articles/LeakyAbstractions.html
[8]http://codingdelight.com/2014/01/16/favor-composition-over-inheritance-part-1/

# Blooper Reel




*The Stretchy Knight: Attempting to perform skinned mesh animations without initializing bone transformation data.*

*Mouth Man: Attempting to render Batman model without allocating texture buffers. Texture buffer 0 is magical, and is always available to render whichever texture was copied last.*