Game Creation using *OGRE* – Object-Oriented Graphics Rendering Engine

Tom Dawson

Introduction

The Open-Oriented Graphics Rendering Engine, henceforth referred to as *OGRE*, is an open-source graphics rendering engine that allows programmers to work in both OpenGL and Direct3D through a straight forward and generic series of interfaces. It is fully extensible, and has been wrapped into many languages; in its pure form, however, it is written in C++. It is relatively easy to modify the underlying engine, and indeed a plugin system exists to allow users to modify or swap many areas of the codebase. The open source nature of the code allows for it to be changed on a per-project basis, if various modifications to the original source are required. In short, it is a viable platform for development upon.

The following paper discusses an attempt to use this underlying engine as the basis for a game engine; *OGRE* itself is purely a means of displaying and managing graphics, though like many rendering engines of its type the steps needed to apply it as a game engine are well defined. The paper will cover both the development stages of a simple demonstration-type game, and the various techniques that can be implemented throughout *OGRE* to achieve similar results.

In terms of the game to develop, inspiration was drawn from *WipE'out* (Psygnosis, 1995), a Playstation game (now also updated to Playstation 3 as *Wipeout HD* (SCEE, 2008)) featuring “hovercraft” of a type racing around a

futuristic track. The high speed and strange track arrangements of the game would provide an interesting number of technical challenges to replicate, especially as it was decided to not use static tracks – that is, that the tracks used within the demo game would be variable, and editable within the game engine.

Initial design ideas based the art style and track generation style around the manipulation of vectors. As such, and as part homage to the capitalisation of *WipE'out*'s title, the demo game was tentatively named *VecTRacer*.

Previous Work

In making a game inspired by a previous piece, the first and most obvious area to examine is the previous piece itself. Besides the obvious parts involved in any racing game (a number of laps, timing, and the like), a number of assumptions can be made when looking at *Wipeout HD*; the vehicles are constrained to the track in some way, allowing them to cling to it even when the track is perpendicular to the ground, or even inverted. Similarly, the vehicles do not come into contact with the track in a normal situation. This suggests that there is some sort of spring-like force acting between the track and the vehicle, much like the suspension between a vehicle and the ground (assuming gravity is always downwards relative to the vehicle).

An option exists to allow for automatic steering away from the edge of the track, suggesting that either a series of rays are cast from the vehicles to detect obstacle distance, or the edges are stored in some other way. The AI is able to navigate the track smoothly, and to cope with corners. This suggests that some information about the track is stored in the form of a spline, which when interpolated along can be used to pull out important details – such as the width, the speed that is required at that particular moment, and other such data. Indeed, the curvature of a spline can generally be used to dictate the required speed at any particular time, though some look-ahead calculations would be required in a simulation that uses real braking. This is rather dependent on whether the spline is nonuniform or equalised (see Lowe,

2004) . Such a spline could be treated slightly differently by each vehicle using it, allowing for smooth and non-repetitive AI behaviour. The possibility also exits, of course, that multiple splines are used on a single track to provide the AI with a choice on a per-corner basis. These are more than likely generated by the level design team rather than on the fly, although the presence of a "ghost" craft flying the player's old route suggests that the ability is there to create spline content at run time.

Another contemporary game that appears to use a spline to provide a track is *Audiosurf*, a game also featuring procedurally generated tracks, in this case generated from an input audio track. The spline following behaviour is more profound in *Audiosurf* (Fitterer, 2008) than as in *WipE'out*, as the player is constrained to only moving on the XY axis relative to the spline (the axes used throughout this paper will be talked about in terms of *OGRE*; *OGRE* uses the negative Z axis as "into the screen"). The idea of procedurally generated tracks being created along a spline is interesting, and led to further exploration in that area in the course of the project. The aim of *VecTRacer* was not, however, to utilise music generated tracks, with the focus instead being on player creation.

WipE'out has many thematic descendants. The indie game *Aftershock* (Liquid Rock Games, 2010) is of particular note, as it also utilises *OGRE* as a graphical engine. The developers update their development log frequently, with many of their findings being similar to those discussed in this paper.

Technical Background

On Splines:

OGRE itself boasts an implementation of a Catmull-Rom (See: Twigg, 2003) spline, a form of Hermite spline. Catmull-Rom splines are often used in graphics engines, and are particularly favoured due to the fact that the spline will pass through every control point – compare this to a bezier spline, which will generally not. Catmull-Rom splines are uniform splines, however, which in certain regards can be a disadvantage. In a uniform spline, the velocity between each control point is the same, regardless of the distance between the two points. This means that interpolating between two control points will produce a large disparity in distance and curve velocity when comparing two close together control points, and two that are far apart; this is not ideal for calculating a smoothly flowing track.

To counteract this, it is possible to equalise the spline. In an equalised spline, each control point is the same distance away from its neighbours. This minimises the effects of velocity changes in the corners, and thus smooths the corners of the curve. In very loose pseudocode:

```
for each (point in spline as X):
        check distance from X to next point
        if distance is less than required then get next point and check again.
        if distance is greater than required then store checked point as Y

interpolate halfway between Y and its own next point (Z).

check distance from X to interpolated point (A)
        if distance is less than required, interpolate halfway from A to Z as new A
        if distance is greater than required, interpolate from Y to A as new A
        if distance is within tolerance, insert A as a new point in the new spline

continue from A as X
```

This results in an equalised spline with equidistant points along its length.

On Physics:

OGRE, being a rendering engine, does not have any native physics support built in. Indeed, it only has very limited collision detection, with additional plugins usually required to do more than very basic ray queries. Fortunately, due to the presence of open source coders this is easily rectifiable through a few plugin systems. One such plugin is *OgreBullet*, a wrapper between the open source physics library *Bullet* and *OGRE*. Whilst *Bullet* can be accessed from directly inside any develop environment with the right libraries, *OgreBullet* allows a programmer to use the two libraries together relatively seamlessly, as it will convert between *OGRE*'s internal data structures and *Bullet*'s, leaving the developer free to deal with other concerns.

Bullet is a very complete physics library, and provides a developer with a number of useful interfaces. As mentioned above, there is a requirement in a game of this type to model a set of suspension between a vehicle and the surface upon which it rides. Bullet itself contains a class designed to represent

a vehicle (some details of which can be found in Maddock, 2010). Whilst the documentation on *Bullet* is relatively sparse compared to *OGRE*'s, the available resources more than outweigh the disadvantage.

On the Engine:

OGRE is very well suited to game development, with only a minimal amount of start up time actually needed to be invested to provide a workable test game framework. “Out of the box”, it comes with a powerful resource manager, scene manager and frame listener functionality, which is of course completely customisable. *OGRE* scenes are typically formed of a hierarchy of scene nodes underneath a single root node inside the manager. The advantage of this hierarchy is that not only will transformation and orientation details will propagate through the tree to each leaf node (if required – this functionality can be disabled on a per-node basis), but the tree can also be used to perform visual culling operations on areas outside of the current camera's viewport.

OGRE handles actual entities (or moveable objects) separately to scene nodes; that is, an entity can exist independently of a scene node. The entity will only become visible when attached to a scene node, and can be detached and moved to another node at will.

Implementation

The core intent of this project was to highlight the possibility of creating a demonstrable racing game running within the *OGRE* engine, with a track that is editable by the player. The track was to resemble as much as possible that that would be found in *WipE'out*.

Obviously, this would require some visual aesthetics to give the impression of a fully functional demo, though the functionality and readability of the code was valued more than the visual aspect. Readable code is always aimed for, though as projects progress and time becomes more pressing this is often the first thing to fail.

Audio analysis was toyed with, but ultimately discarded in favour of keeping the core functionality, as was any multiplayer aspect. Such aspects, while nice in theory are often best left until later stages of development, when the core functionality has already been laid in place. This is especially true in the case of any multiplayer, as when the core of the game changes entirely the multiplayer must by necessity change with it – not to mention the required extra networking code that is required to create anything beyond multiple players on a single keyboard.

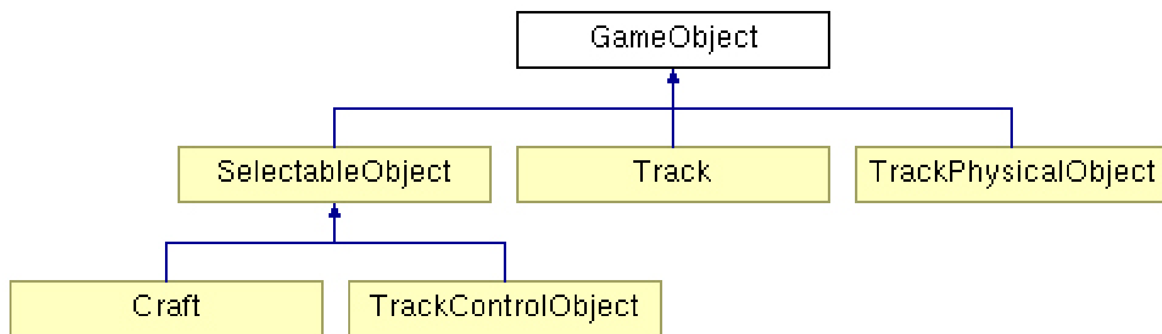
The following section will be divided into a number of sub-sections, dealing with the three main areas that were explored within the actual course of the project; the engine itself, the track generation, and the physical aspects of the player craft. To a lesser degree, the pipeline of adding content to the engine shall be discussed.

On the Engine:

The first step of setting *OGRE* up as a game engine is much the same as setting it up for any other sort of application. A generic Application class is created to store the various managers – such as camera (*CameraManager.h*), object (*ObjectManager.h*), physics managers (*PhysicsManager.h*), and an input handler (*InputHandler.h*). The Application itself is a singleton, a class which is instantiated at the start of the program and provides a static method to return a pointer to itself. This allows all the managers to access each other through the central "hub" of the Application (*Application.h*).

Outside the *OGRE* framework, a hierarchy of *GameObject*s (*GameObject.h*)was created. A game object in this instance stores both its own entity and scene node information, along with various pieces of information to deal with moving the actual object. Inheriting from this is the selectable object class. This is essentially the same as a game object, with the added methods that allow it to be selectable via the *InputHandler* class. These two classes are stored and accessed through the *ObjectManager*, which every frame attempts to call a *move()* function on these objects. In the most basic use of the *move()*

function, the object attempts to move along its own vector if it is not set to 0. In some derived classes further down the tree, however, this *move()* function is used to take advantage of the fact that it its called every frame, using it for updates. An example of this is the track object using it to update the GUI with the current lap timings on a per-frame basis.



TrackPhysicalObject and *TrackControlObject* are thematically similar classes, and share many of the same methods; however, they are kept separate to ensure clashes of interest do not occur.

OGRE provides methods on a frame listener to run both before and after the rendering of a frame. These are accessed to force the various managers to update at the same time – or at least to attempt an update, as all of them are given the time since the last frame. This can be seen in *FrameListener::frameRenderingQueued*, which handles input, camera movement, object movement and the physics step.

The GUI is drawn in a second viewport, and blended over the top of the

full 3D viewport. The input handler is configured to switch between interacting with the two, depending on the current application state – starting up, running, or shutting down – and the current game state. *OGRE* does not allow ray checking on billboard objects, which form the text of the GUI. It is a simple matter to add a 3D cube scaled to the appropriate size behind the text object, though, which if linked correctly allows for mouse interaction with the menu.

The *InputHandler* uses *OGRE*'s ability to store an *OGRE::Any* pointer on many of its internal objects. This pointer, as the name suggests, may point to any other data type. It is used in conjunction with a ray scene query (provided by *OGRE*) from the mouse position into viewport space to select an object. With dynamic recasting, the object can then be interacted with, by calling various methods on the GUI handler in the case of a GUI element, or by directly accessing methods on the selectable object, as in when interacting with the player's craft.

Due to the use of the *OGRE::Any* pointer type, some cast sanity checking takes place. This is a simple check of the *m_type* variable present on all *GameObject*s. In most cases, *OGRE*'s casting does not fail. However, the player *Craft (Craft.h)* object – for collision detection purposes – uses *Bullet*'s own internal pointer storage and recasting. Whilst a more elegant solution could no doubt be found, checking types like this is always a sensible precaution.

On the track generation:

Generating the track was perhaps one of the most in depth parts of the project, as the track would have to undergo the spline normalisation discussed above, have geometry generated from the spline, and ultimately be set to the physics engine as a mesh.

The track generation process begins with the construction of a spline. For ease of use, the first points of a spline always form a straight line. This provides a natural “home straight” for the finish line to lie upon, and prevents issues with geometry clipping in this area. The next points of the spline are arranged in a set order, and the player is able to manipulate them to a certain degree. The player is not able to manipulate the straight line points immediately surrounding the finish line, for the same reason mentioned above. Once the player is happy with the spline, it must go through the generation phase.

This phase is the step that consumes the most time. When the track is equalised, a number of *trackPhysicalObjects* are created, that coincide with the interpolation points that are created. These physical objects are used as markers for each geometry section. The geometry itself is further broken down into a series of vertices by interpolating between these markers. Each interpolation provides a directional vector, alongside which is the presence of the overall directional vector from each physical object. The cross product of these vectors with the world up vector (*Ogre::Vector3::UNIT_Y*) would then

produce a line parallel with the ground at each interpolation step. Whilst this would be sufficient in many cases, an entirely flat track is not visually interesting, nor does it conform to the *WipE'out* style.

The next stage, therefore, is to generate some sort of banking to the corners, using the positions of the nodes being checked. This was done by taking the incoming direction vector of the first node and finding the angle to the outgoing direction vector of the second. With constraint to a maximum banking angle (to prevent banking of over 90 degrees and a completely sideways track), this angle could be used as an average up vector for the track section. In fact, the angle is constrained to around five degrees, as even this is a large banking angle when seen from the point of view of a vehicle.

The previous cross product calculation could then be modified to use this average up vector instead – or, as ultimately resulted, an interpolated up vector to match the interpolated position along the track spline. With some careful checking to ensure the banking was sane (based on the world position of the respective nodes), a generally smooth curve was created.

A problem still lies in this method of banking generation. The cross product on a banking corner is used to extend the track both directions from the spline's location, which results in part of the track lying in the negative Y direction. In an attempt to remedy this, and ensure the track never loses height in a bank and only gains it, the whole segment was shifted up in Y until the lowest point lies upon the XZ plane, at 0 Y. This did not have the desired

result, however, as it would exacerbate minor changes in the geometry and create large shifts of track angle. After much deliberation, the track was left as is, and allowed to dip into negative Y.

As it may have become apparent, all of this geometry drawn on the fly. For minor changes such geometry creation is no problem, however when generating an entire track the calculations rapidly become rather expensive. Alongside that, when the player is actually racing on the track there is no longer any requirement for the track to be editable. The class used by *OGRE* to manually edit objects in this manner is the *ManualObject* class. Fortunately, this class provides a method to convert the *ManualObject* to a mesh, which can then be used by a standard entity. At this point, the *ManualObject* can be deleted to save memory.

As a further optimisation, which became required on larger tracks, these individual meshes can be lumped into their own *StaticGeometry* bins. *StaticGeometry* is an optimised class of *OGRE*'s, which is meant to deal with (as the name suggests) static geometry, which is not expected to move during its lifetime. It also aids the scene manager in culling unseen geometry.

Before the meshes are deleted (to remain only in the *StaticGeometry* bin), they are also passed to Bullet to create a collision mesh from, allowing the player vehicle to interact with them.

One important thing to note at this stage is the Visitor class. This class,

derived from *Ogre*'s *Renderable::visitor*, is called upon occasion to “visit” the static geometry bin, via the *visitRenderables* method. The *Visitor* class has its own *visit()* method, which the *visitRenderables* method calls on every renderable in the bin. Whilst this does not sound essential, it allows for the updating of custom variables on the renderables – which are in turn used by the shaders. This was put to use to pipe the player position to the fragment shader used in the track walls, and many of the track objects.

Currently, no sanity checking is performed on the tracks. It's very possible to create a track that is impossible to race upon, which is a fairly major issue.

On the player craft:

Initially, the player craft was designed explicitly to not use the *Bullet::raycastVehicle* class, as the *raycastVehicle* did not seem like the ideal solution. The initial idea was to place a spring constraint between the vehicle object and the track plane, and to use the inverse up vector (derived from the geometry as above, and accessible to the craft) as a gravity force rather than *Bullet*'s standard gravity. *Bullet* operates entirely in forces, and as such gravity is simply a downward force in the world Y; applying this same force in positive Y rather than negative serves to negate it entirely. The up vector from the track would also be applied as a constraint to the craft's own rotation, keeping it upright relative to the track rather than the world. However, in tests the differing gravity force would still cause the craft to slide in a certain direction

when sitting on relatively “flat” terrain, regardless of the world orientation. Additionally, the upright constraint did not give pleasing results when coupled with only one spring.

As such, investigations were made into the *raycastVehicle*. A raycast vehicle, as the name suggests, uses a number of rays to simulate the suspension of a vehicle, much like the spring constraint used on the simpler model would. Alongside this, however, a “wheel” is modelled at one end of the suspension, with accompanying friction values. It is possible too to apply differing torques and steering values to each wheel independently, although this was ignored in favour of directly applying force to the vehicle.

Initial tests with the vehicle found it to be unstable in corners, even with a roll influence modifier applied (a modifier which essentially cheats the physics system, and makes the vehicle less likely to roll). There were a number of causes of this; the vehicle had a high centre of gravity, as by virtue of being a hovercraft it had a very long set of suspension to give the “floating” feel. *Bullet* does not directly allow for the centre of mass to be manipulated, but uses the origin of the model instead. By editing the model directly and shifting the origin into a more useful location, the centre of mass can be “altered”.

The upright constraint that was previously used with the spring based model was also implemented on the vehicle model. The vehicle model, however, does not respond well to being constrained in such a way; if the free angles of rotation in X and Z are too low, the physics solver will encounter

errors and behave in an unexpected way, often catapulting the vehicle large distances. As a side note, bullet does not allow completely free rotation about the Y axis, in an effort to avoid gimbal lock. To circumvent this, the reference frames can be altered to use X as the local Y axis; this is generally only a problem when creating constraints of this type, although gimbal lock is certainly never desirable.

Another assumption that caused problems with the bullet vehicle model is that the ray cast suspension of a rigid body would not collide with the same rigid body that it is meant to act upon; this is not, however, the case. If the suspension begins above the rigid body itself, it may collide with it and promptly exact a force upon its connection point, often sending the vehicle up into the air with no visible cause. If the suspension is within the rigid body or below it, however, the problem obviously does not occur.

To give the feel of a hovercraft, force isn't directly applied to the vehicle via the wheels, nor is any steering influence. Instead, a large amount of thrust (a force) is applied at the rear of the vehicle, and turning is achieved by a much smaller amount being applied to the front of the vehicle in the relevant direction. The wheels are also set to have a low amount of friction. A damping amount is added to the rigid body to act as air friction, and a force is also applied against the vehicle's linear velocity when the back button is pressed to act as a braking force. To prevent the vehicle flying, thrust is only allowed to apply when within a certain distance of the track.

For pure visual effect, a grid is projected downwards from the craft. This takes the form of an orthographic projection frustum, which does not inherit the orientation of the craft. OGRE allows decals to be projected in this way onto any texture required, as part of a dynamic render pass.

As can be seen from the demo files, the physics solver still encounters a number of collision issues. On particularly complex tracks it is possible for the “wheels” to fall through the geometry, or become lodged. At this time, it is unclear whether this is due to the construction of the geometry itself or because of a setting within the raycast vehicle's suspension.

On the pipeline:

For those considering using *OGRE* for a graphics project, it should be noted that the .mesh format it uses is unique to the program. Exporters have been written for many major 3D applications, but not all are perfect. At the time of writing, attempts to compile an exporter for any 64 bit version of *Maya* have been met with failure. As such, the pipeline for adding models to this project consisted of creation in *Maya*, exporting to an .obj, and opening again in a program for which a working exporter could be found – such as *Blender*. Of course, if learning time or resources are not an issue a compatible program can be used from the start.

Conclusion

The project seems to be an fairly adequate representation of how swiftly a working game engine can be implemented in *OGRE*. Due to the extensible nature of the engine, it is very easy to plug in extra components as time goes on, and the object oriented build makes understanding the majority of the components of the engine simple, even when encountering them for the first time. In terms of the original idea, that of creating a user editable *WipE'out* style game, the project is well on track. The actual playability of the game is in question, however, due to the presence of a number of limiting bugs.

Bugs, areas of concern, and solutions

As mentioned above, the track is a major area of concern. Due to the high velocity displayed in the spline, some corners are able to double back on themselves and create an impossible corner; even a corner of 90 degrees causes major visual artefacts and is hard to navigate. Any series of corners where the track's planes intersect with each other also cause problems, as the track side barriers will appear where the player would not expect them to.

An obvious first step would be to implement a different type of spline for the corners; Barrera (2005) describes a Hermite curve that seeks the minimal amount of acceleration and thus turn rate for a corner, whilst Lowe (2004) describes how rounded nonuniform splines can be used to smooth a track.

Either implementation would aid the problem, though likely not solve it entirely.

Increasing the amount of geometry present in each track segment would provide a smoother visual look to the segment as well as aiding the physics engine, albeit with a trade off against speed.

Most importantly, intersection checking both within and between individual track segments would allow for the prevention of impossible segments; again, the trade off here comes at the cost of generation speed, although it can be seen that the current implementation is still sufficiently fast.

Finally, as it currently stands it is entirely possible to create a looping section of track that intersects with and entirely crosses itself. This will create a barrier extending across the entire width of the track, rendering it impassable. An initial idea to limit this was with the implementation of a vertical element to the track. If a track can loop above or underneath another track, the problem is instantly nullified. Of course, this involves yet more pre-processing on the track during the building phase. As such, the building phase would necessitate being split into a series of discrete loading chunks. Whilst this would not improve speed in any way, it would provide the player with some visual input as to what is actually occuring at each stage of the process, rather than simply waiting with a blank or stalled screen.

The physics solvers on the player craft too cause issues. Whether this is due to unfamiliarity with Bullet's physics solvers or an artefact from the generation of the track remains to be seen. In an ideal situation, the vehicle solver would not be needed and the simple spring idea described above would be sufficient. To fully implement the simple spring idea a greater deal of interaction would be required between Bullet and the application itself. For example, instead of attempting to pull out the pre-calculated up vector on each intersection of ray and track, the normal vector of the intersected plane could be used instead. This would not only give an aesthetically pleasing result, but would be more accurate in cases of extreme track banking. With only one contact point between vehicle and track there would also be less calculations required per frame.

Further development

For further development, it is obvious that the issues with both craft and track are priorities. Once these problems are dealt with and the core functionality is enabled, development is free to expand in any direction. An obvious first choice is to extend the track generation process by allowing the saving and loading of tracks once edited, allowing the same track to be raced upon multiple times. This too would give the presence of lap timers some meaning, as with unique (or nearly unique) tracks being generated each time comparing times is meaningless.

An initial idea for the project at its inception was for the track to be

editable by multiple users at once, or even in real time. A recent game, *Split Second*, allows the players to interact with the track on a per-race basis. In the case of *Split Second* this interaction comes from scripted events that permanently alter the track; in the case of *VecTRacer*, the intent was to allow players to alter the track in a “pre-game” state, possibly as an alternative to upgrading their craft.

Further extensions would be to expand the way tracks are generated. Already mentioned was the idea that tracks involve some sort of vertical movement; if tracks were able to split and rejoin, or interact with pregenerated geometry interesting styles of play and interaction may occur.

Finally, the actual aesthetics of the game are in need of improvement. Whilst this was not a major aim of the project at the time of writing, graphical and audio improvements always serve to make a product more appealing to the end user.

References

Papers

Barrera, T. 2005. Minimal Acceleration Hermite Curves. *In:* Pallister, K., ed. *Game Programming Gems 5*. Hingham, MA: Charles River Media, Inc., 225-231

Lowe, T. 2004. Nonuniform Splines. *In*: Kirmse, A., ed. *Game Programming Gems 4*. Hingham, MA: Charles River Media, Inc., 171-181

Twigg, C., 2003. *Catmull-Rom Splines*. Available from: http://www.cs.cmu.edu/~fp/courses/graphics/asst5/catmullRom.pdf [Accessed 20 August 2010]

Maddock, K., 2010. *Vehicle Simulation With Bullet*. Available from: https://docs.google.com/Doc?docid=0AXVUZ5xw6XpKZGNuZG56a3FfMzU0Z2NyZnF4Zmo&hl=en

Libraries

Coumans, E., 2010. *Bullet* (2.76) [library]. Available from: www.bulletphysics.com [Accessed 20 August 2010]

The Ogre Team. *OGRE* (1.7.1 [Cthugha]) [library]. Available from: www.ogre3d.org [Accessed 20 August 2010]

Kuranes, T., 2010. *OgreBullet* [library]. Available from: http://www.ogre3d.org/developers/addons [Accessed 20 August 2010]

Games

Liquid Rock Games, 2010. *Aftershock* [computer game]. Not yet published. Available from: http://www.liquidrockgames.com/

Fitterer, D. 2008. *Audiosurf* [computer game]. Online: Steam.

Black Rock Studio, 2010. *Split Second: Velocity* [computer game]. CA: Disney Interactive Studios

Psygnosis, 1995. *Wipeout (WipE'out)* [computer game]. Liverpool: Psygnosis.

SCE Studio Liverpool, 2008. *Wipeout HD* [computer game]. London: SCEE