# HURA Game Production Document

(MSc Computer Animation and Visual Effects, Masters Thesis)

By Ian Thompson

## Table of Contents

HURA Game Production Document ........ 1

Outline ........ 4
Project Brief ........ 4
Research and Design ........ 4
Roles ........ 4

Unity Overview ........ 5
UniWii Plugin ........ 5
Asset Management ........ 5
Scripting Languages ........ 6
Behaviour Model ........ 6
Multithreading ........ 7

Animation Pipeline ........ 7
Rigging Quadrupeds ........ 8
*Separation of Controls ........ 8*
*Problems with Three-bone Chains ........ 8*
*Expressions ........ 9*
*Grouping and Synoptic ........ 9*
Animation Export Process ........ 10
*Plotting and Reduction ........ 10*
*Rig Reduction ........ 10*
*Final Export ........ 11*
Animation Import ........ 11

Asset Management ........ 12
Version Control ........ 12
Unity Asset Management ........ 13

Tools Development ........ 13
Spawn Points ........ 13
Checkpoints ........ 14
Constraints ........ 14
*Bézier Path Constraint ........ 15*
Collision Mesh Replacement ........ 15

Gameplay Programming ........ 16
Camera Controller ........ 16
Player Controller ........ 16
Boost Mechanism ........ 17
*Token Pick-up ........ 17*
Track Boundary Checking ........ 18
Competitor Instantiation ........ 18

Animation ........ 20
Blending ........ 20

Speed Adjusting & Limiting ........ 20
Synchronisation ........ 21

**Artificially Intelligent Opponents ........ 21**
AI Network Nodes ........ 21

**Human Interface ........ 22**
Wii Remotes ........ 23

**Graphical User Interface ........ 23**

**Design Alterations ........ 24**

**Testing ........ 25**

**Conclusion ........ 25**

**Credits ........ 25**
*Core Team: ........ 25*
*Audio: ........ 25*
*Testers: ........ 25*

**Figures ........ 26**

**References ........ 27**
*Software ........ 27*
*Computer Games ........ 27*
*Articles ........ 27*
*Unpublished ........ 27*
*Websites ........ 27*

**Appendices ........ 28**
Appendix A: Class Inheritance ........ 28

## Outline

The project has been developed in two stages: preproduction – developing the concepts and designs behind the game; and production – the actual asset creation and game development.

### Project Brief

The core concept was to develop a "pick-up-and-play" game utilising the Nintendo Wii motion controls. The game itself was to be a racing game based around a fictional 1950s English village where farmyard animals are raced in an annual event. The companion Game Design Document should be consulted for further information.

Rather than an in depth research and development exercise this project comprises a focus on planning, management and production. As the large project this was, several areas were covered and many problems faced along the way. How these problems were overcome in a team-oriented production environment was important to the success of the project.

### Research and Design

During the preproduction phase of development, the entire game was meticulously designed. Every aspect, from character and environment design through to the boost mechanism, was discussed and firm decisions made at this early stage. This laid an excellent platform to allow the game in mind to be quickly and efficiently developed and gave a clear and consistent picture of the final product. Of course, there were changes made as alternatives were discussed and further possibilities were explored but the central concepts have remained throughout. For a discussion on the changes that were made, see the section Design Alterations.

Previously released games in the same genre were consulted for comparison. Particularly racing games utilising the Wii remote and its capabilities as a driving controller. Such games included Mario Kart (Nintendo EAD, 2008) and Excite truck (Monster Games, 2007) along side many others that were looked at during preproduction (Griffiths et al, 2009).

### Roles

During the development of the game, the author took on all technically based roles:

- Gameplay Programmer
- AI Programmer
- User Interface and Input Controls
- Tools Development
- Animation Rigger
- Pipeline and Technical Director

The other members of the team (Sophie Shaw and John Griffiths) took on the artistic roles:

- Character Modelling
- Environment Modelling
- UV and Texturing
- Level Design and Production
- Character Animation
- Enveloping and Weight Painting
- Set Dressing

Other roles included music composition, sound effect production and testing. Kirstie Hewlett and Matt Kennedy took on the former two roles respectively, whilst testing was carried out over the second half of the project by a whole team of volunteer testers. For a full list of people and their involvement in the project, see the section entitled Credits. Testing is covered in more detail in the Testing section.

## Unity Overview

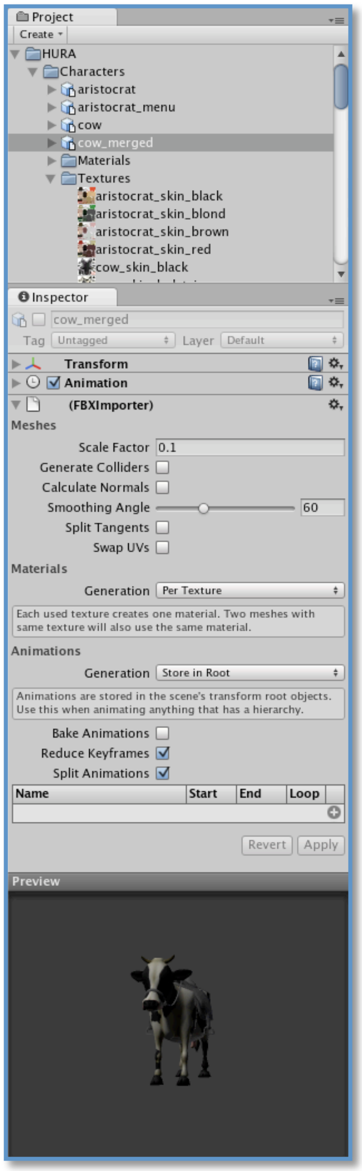
The engine of choice for this project was Unity, a fully-fledged game engine and development environment, primarily chosen for its Wii integration. Unfortunately, with the high costs[1] imposed when developing and publishing to the Wii platform we decided to take an alternative approach. Bearing in mind that Unity has the capability to publish to the Wii we had only to deal with connecting the Wii motion controllers to our development PC. This was a case of obtaining the correct software and utilising the Bluetooth signal from the Wii remote. We found this worked best on the iMac, which is also Unity's native platform[2].

### UniWii Plugin

Early investigation lead to the discovery of a Unity plugin called UniWii. This plugin provided the necessary functions to poll and interface with the Wii remotes by utilising code from the open-source project, DarwiinRemote.

### Asset Management

Unity maintains all assets under one hierarchy, making organisation relatively straightforward. It imports each



cow_merged
Tag: Untagged  Layer: Default

Meshes
Scale Factor: 0.1
Generate Colliders: ☐
Calculate Normals: ☐
Smoothing Angle: 60
Split Tangents: ☐
Swap UVs: ☐

Materials
Generation: Per Texture
Each used texture creates one material. Two meshes with same texture will also use the same material.

Animations
Generation: Store in Root
Animations are stored in the scene's transform root objects. Use this when animating anything that has a hierarchy.
Bake Animations: ☐
Reduce Keyframes: ✓
Split Animations: ✓

| Name | Start | End | Loop |
|---|---|---|---|

Figure 1 – Inspector in Unity

[1] Unity licences start at $15,000 for WiiWare distribution (Unity, 2009 a). On top of this is the cost of purchasing a Wii DevKit and becoming a licensed developer.

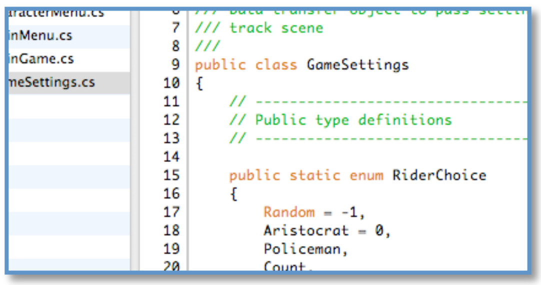
[2] Unity was recently released for Windows (Unity, 2009 a), allowing for much easier collaboration and integration with our XSI pipeline.

asset from a variety of supported formats and hides the process, exposing only certain, specific options. Each asset therefore, like all objects in Unity, has a property page. Everything is treated as an asset so scripts also have their own property page. Property pages appear in the Inspector.

Public fields of any class derived from `MonoBehaviour` are accessible directly from the Inspector. This is most useful for linking scripts to other objects at design-time so that they need not be sought during execution. It also allows assets to be easily linked into scripts. For example, the `UIButton` class has a public Texture field to allow the script to be passed an image asset. Once running, this script then has complete access to the texture instance and can draw it to the screen. See the section on User Interface for details of how the same texture may be used at design time to lay out the UI components.

## Scripting Languages

Unity supports three scripting languages: JavaScript, Boo and C# (Unity, 2009 b). There is little performance difference between these and as such the choice was left open. Having prior experience with both JavaScript and Python (on which Boo is based), the choice was made to use C# to expand the author's skill set. It is also possible to mix and match scripts in different languages since they share the same API under the Novell (2009) sponsored Mono project, so the risks of learning a new language were snuffed. C# is the language of choice for XNA and many companies are using Mono, as Unity does, for their scripting needs (Novell, 2009).



```
/// track scene
///
public class GameSettings
{
    // -----------------------------------
    // Public type definitions
    // -----------------------------------

    public static enum RiderChoice
    {
        Random = -1,
        Aristocrat = 0,
        Policeman,
```

Figure 2 - Unitron script editor

## Behaviour Model

Scripts sit atop `GameObject`s as behaviours for that object. Where the script controls the underlying object, be it a camera or player, this model makes perfect sense. However, it means that for a script to respond to events and it must be attached to an object in the game, which in turn has transform data and so on. There is no central script, no main method. This has lead to a set of single instance classes to hold data for a more global context and to which the behaviours have access at all times.

The main "global" singularity classes for this project have become:

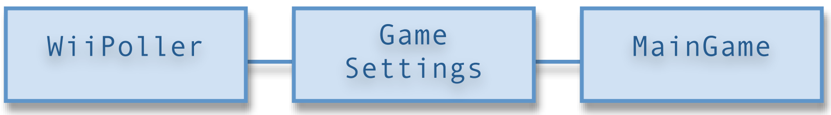


Figure 3 – Singularity classes

- `WiiPoller` – this works with the UniWii plugin to provide constant data from a set of Wii remotes.
- `GameSettings` – this is a persistent object allowing settings to be passed from the menu scenes into each track/level scene.
- `MainGame` – an object recreated by each track scene to set up and maintain game properties, such as the `Competitor` instances in the race.

### Multithreading

While developing the singularity classes and others where data were shared among different objects it was a concern that concurrent access would cause problems. However, it was discovered that Unity was not written to be thread-safe and as such uses only a single game loop. Since each object has its own behaviour, independent of other objects, the structure would lend itself well to multithreading.

Although allowing the code to be much simpler by keeping a single game loop, today's computers and games consoles often have multiple cores, which would benefit greatly from a multithreaded solution.

> *"In the last couple of years improvements in single processor hardware have approached physical limits and performance gains have slowed to become incremental. As a consequence, improvements in game engine performance have also become incremental. Currently, hardware manufacturers are shifting to dual and multi-core processor architectures, and the latest game consoles also feature multiple processors. This presents a challenge to game engine developers because of the unfamiliarity and complexity of concurrent programming. The next generation of game engines must address the issues of concurrency if they are to take advantage of the new hardware."*
>
> Tulip et al. (2006)

Unfortunately, Unity was established before this, with its first release arriving in 2005 (Unity, 2008), and the threading model has not changed since then. However, our initial target platform was the Wii, with its single-core IBM PowerPC processor. Despite changing to develop on a PC with multiple cores, we can gauge the performance better for Wii due to Unity's single-threaded architecture.

## Animation Pipeline

From the initial tests done prior to production (see the Game Design Document), a pipeline could be established that would allow the characters, modelled, rigged and animated in XSI to be exported for use in the game engine, Unity3D.

## Rigging Quadrupeds

Following on from an initial study into rigging for quadrupedal characters, a rig was constructed for our first animal. This ensured the pipeline worked as expected and allowed character animation to commence.

### Separation of Controls

A core design decision was to separate the bone system entirely from the control system. It was decided that the skeleton to be exported would be constrained to a series of controls, the controls exclusively having keys set. This meant all animation data was on the control system and the bones were affected by constraints to these animated controls alone.

Having no animation data on the bones meant that during export we could bake out the animation to these bones, remove the constraints and delete the control objects. Thus leaving a much-simplified rig to affect geometry in the game.

It was later discovered that more of each bone chain could be removed while maintaining animation. A rig-reduction script was therefore developed (see Rig Reduction).

### Problems with Three-bone Chains

The first rig contained chains of three bones, driven together by an IK effector. Using the stiffness property of the upper bone it was possible to adjust the behaviour of the limb on the fly. In the initial tests this worked well and provided a very realistic motion for all four limbs of the cow model.

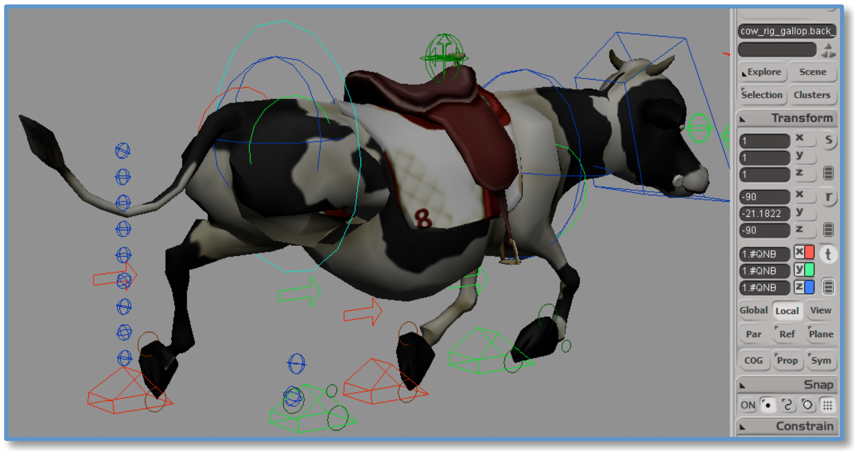


Figure 4 - Disappearing limbs when rotation limits are put in place

During animation, however, it was soon evident that the chain would not reset leaving each limb with a different rest pose.

The first solution was to restrict the rotation of the primary bone by setting its rotation limits. Both the maximum and minimum were set to the same value and constrained (via an expression) to that of a new control object. Initially this solved the problem but the limitations of XSI's IK chains were soon evident as the legs began to shoot off to some phantom location (1.#QNB in XSI terms).

Any new solution at this point would need to minimise its effect on the rig and animations that had been created so far. The bone chain, since it was not animated itself, could be reconstructed as two chains thus giving a single and two-bone chain. From the new controls added for the rotation limits, each hipbone could be mapped directly. The lower, two-bone chain could then use normal IK with its effector in the foot as before. The lesson was learnt not to use more than two bones in an IK chain if full control (and resetting) is required.

Later an alternative IK evaluation method was discovered that does allow bones to be reset to their "preferred angles". This was adopted for all subsequent rigs but it was also decided not to use IK bone chains longer than two bones.

## Expressions

Certain areas of the rigs required special attention. The tail for the cow has a series of bones whose rotations are calculated as the sum of all rotation controls earlier in the chain. This allows the whole chain to coil up from one control and allows this to occur additively at any point along the chain.

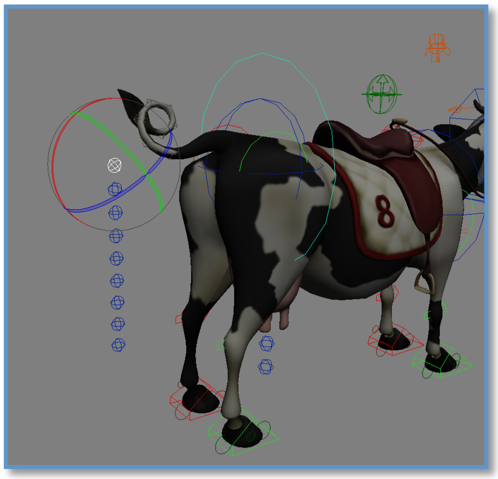


Figure 5 - Tail curl controls

The various spines required particular attention too. The quadrupeds have bones constrained to curves, which are controlled themselves by a set of nulls. The four control points of the Bézier curve are constrained to four nulls, the middle two of which are parented to their respective ends. Thus, by rotating either end the curve twists in an appropriate manner and the bones follow (without changing length as the curve does).

Spine twist was added to the pig rig (a better solution than that which had been found for the cow). First, the primary bone of the chain was rolled by the rear control's x-rotation. By then distributing the front control's x-rotation over the remaining bones' rolls, a correct spine twist was created.

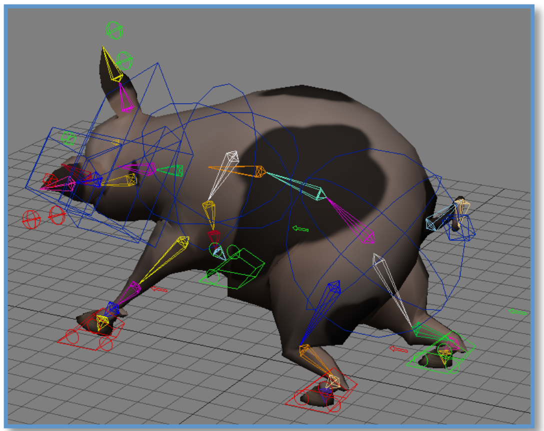


Figure 6 - Pig rig with spine twist

## Grouping and Synoptic

As a tool for animation it was important to provide a consistent interface to the animator. Presenting only the controls and preventing editing of the bones was the first step. Second was to allow these bones to be accessed for weighting. A series of groups was created: bones, controls, joints, etcetera.

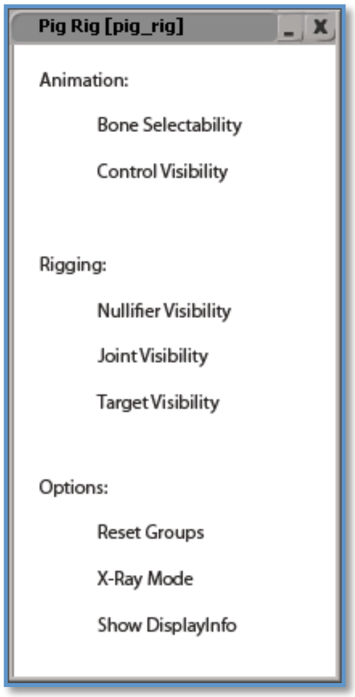


Figure 7 – Rig synoptic

Each group has both visibility and selectability options, which allowed the creation of a synoptic to control these properties and thus set the current toolset for the animator. It was also used during rigging to reduce the amount of information on-screen at any time.

### Animation Export Process

To function in Unity, each character had to be exported into the FBX format. Unfortunately, XSI has limited support for FBX export. To work around this limitation, each animated model was saved out using the dotXSI format, the exporter for which has a number of useful options.

### Plotting and Reduction

When exporting to dotXSI, animation can be plotted onto bones and these bones can be converted to nulls in one step. This essentially reduces the skeleton to its transform data alone. The enveloped geometry is still weighted to the skeleton since only the transform data and hierarchy are required for this. Animations are therefore preserved whilst we are rid of the unnecessary properties associated with bone chains.

Plotting involves taking the animation from the control objects and applying it to the bones per frame so that the bones then contain all the information they require. The controls may then be deleted.

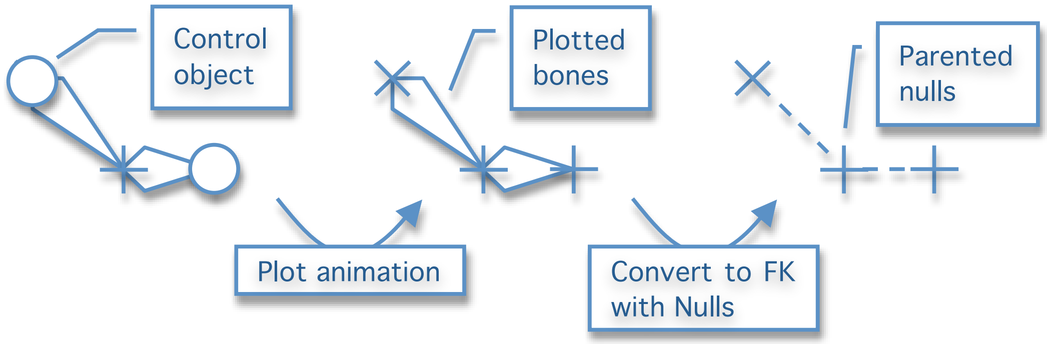


Figure 8 – XSI animation pipeline, first stage

### Rig Reduction

Under inverse kinematics, a bone chain requires a root, bones and a target (called an effector in XSI). Once animation has been baked onto the bones, the root and effector are no longer required (provided the roots are not themselves animated). However this may only be done once the chain has been converted to nulls. Since other objects may be children of the effector they must be repositioned to be children of the last bone in the chain. The first bone of the chain has local rotation but no translation from the root. Copying the root's translation to this bone, and adding its local rotation to all key allows the root to be safely be removed. To do this the bone null is re-parented to the parent of the root.

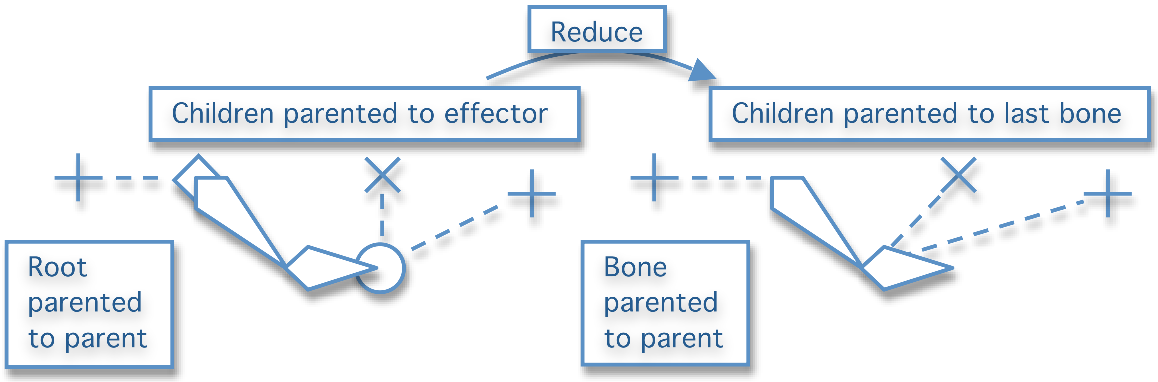


Figure 9 – Rig reduction process

## Final Export

The final step is to export to FBX. Since all that remains at this stage is a mesh (with vertex normals, UVs, and bone weights), and a series of parented nulls weighted to this mesh, the limited FBX support handles the scene and produces a game-compatible animated character.

## Animation Import

To minimise the number of exports, each model had all its various animations placed onto a single timeline. Bringing this into Unity meant splitting the animation back into clips during import. Unity makes this very easy provided the frame numbers are known for each clip (see Figure 7).

Sometimes there would be a discrepancy between the start and end frames of a looping animation clip. Due to the way keys are set in XSI and the export process, the final frame of a loop was always equal to the first so fixing discrepancies usually meant removing either the first or final frame.

Once the clips have been identified in the Inspector they may be accessed by name inside a script by way of the `Animation`, `AnimationClip` and `AnimationState` classes. For more information on how the animations were triggered and blended, see the section, Animation.

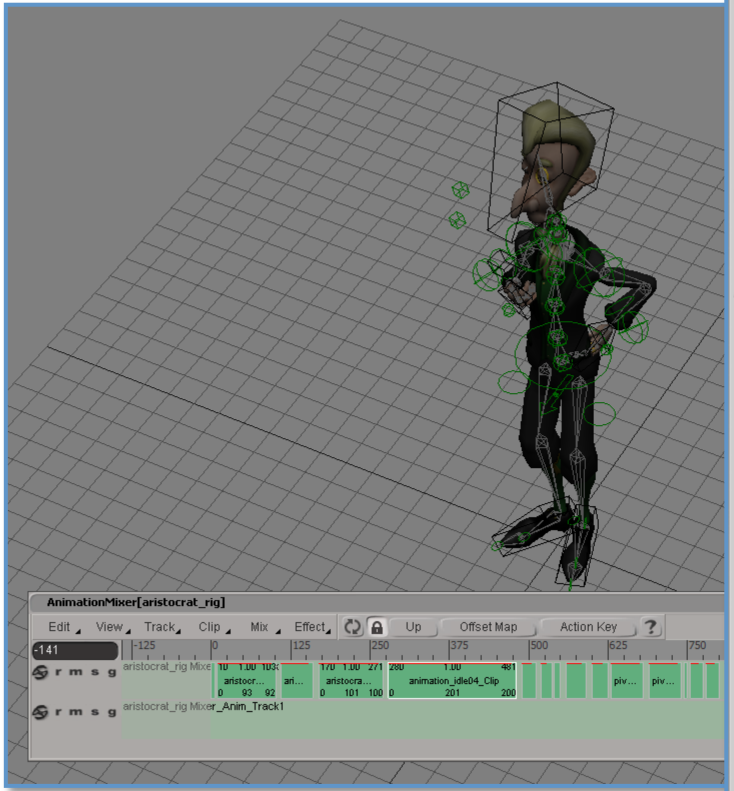
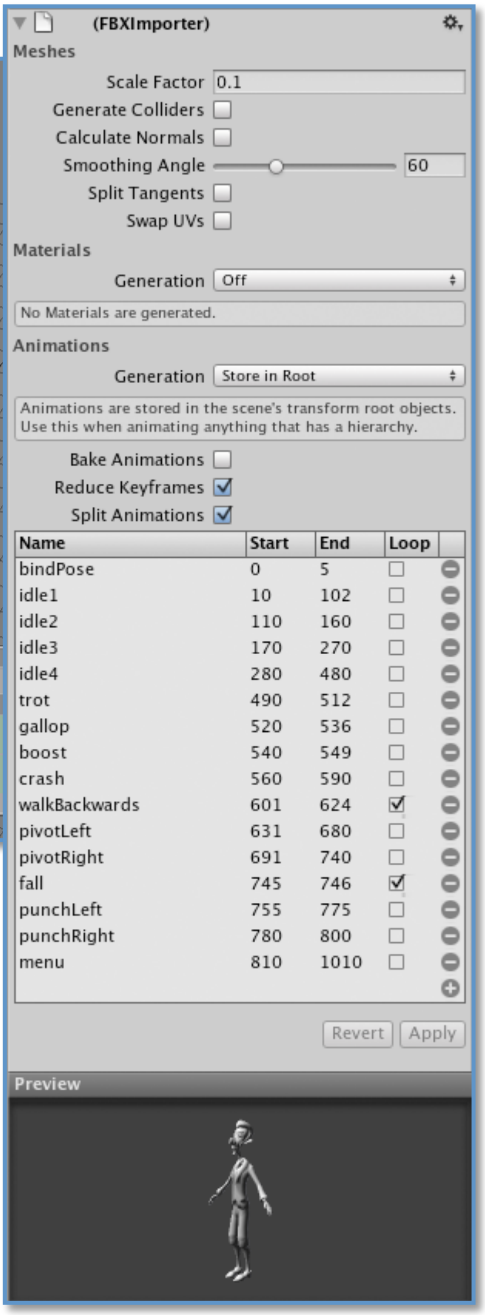


Figure 10 – Animation clips on a single timeline



(FBXImporter)

Meshes

Scale Factor: 0.1
Generate Colliders: ☐
Calculate Normals: ☐
Smoothing Angle: 60
Split Tangents: ☐
Swap UVs: ☐

Materials

Generation: Off

No Materials are generated.

Animations

Generation: Store in Root

Animations are stored in the scene's transform root objects. Use this when animating anything that has a hierarchy.

Bake Animations: ☐
Reduce Keyframes: ✓
Split Animations: ✓

| Name | Start | End | Loop |
|---|---|---|---|
| bindPose | 0 | 5 | ☐ |
| idle1 | 10 | 102 | ☐ |
| idle2 | 110 | 160 | ☐ |
| idle3 | 170 | 270 | ☐ |
| idle4 | 280 | 480 | ☐ |
| trot | 490 | 512 | ☐ |
| gallop | 520 | 536 | ☐ |
| boost | 540 | 549 | ☐ |
| crash | 560 | 590 | ☐ |
| walkBackwards | 601 | 624 | ✓ |
| pivotLeft | 631 | 680 | ☐ |
| pivotRight | 691 | 740 | ☐ |
| fall | 745 | 746 | ✓ |
| punchLeft | 755 | 775 | ☐ |
| punchRight | 780 | 800 | ☐ |
| menu | 810 | 1010 | ☐ |

## Asset Management

### Version Control

Bazaar was the version control system of choice for this project. It works well on several platforms and is relatively simple to understand, requiring knowledge of only a small number of commands.

The server was set up to provide a persistent central repository from which each user could maintain a working copy. The asset creation side was kept separate from the game development to allow delayed application of assets.

Often code would have to be adjusted to support the latest change to a particular asset. It was therefore important that these assets were not updated automatically. To achieve this, changes to assets would appear in the `asset_source` folder (and information about the change in the revision logs). The asset could then be brought into the game engine and the code checked and adjusted as required, committing this as a new revision. Ensuring the assets always moved in this one direction was key to the success of the system.

A more advanced asset management system could aid in this process by queuing assets that require attention before they are applied. It is certainly an interesting problem, and one to look into as a future project.

### Unity Asset Management

Since each `GameObject` may contain multiple scripts and child `GameObject`s it can become a very complex object. It is often necessary to group such an entity for reuse. To achieve this, the parent object may be dragged into the Project manager as a Prefab. Once in this state instances of the prefab may be dragged back into the scene. Each instance is highlighted in blue to show that it is connected to a prefab on disk. Changes to a prefab may be local or applied to all other instances. This allows for a very flexible scene set-up.

A set of assets, prefabs and their associated resources (images, etc.) can be collected together to form a Unity package. These packages tie all dependencies into one neat file and were the means exploited to transfer environment changes into the development scene. Separate scenes were in use during development to ensure that independent work could be carried out whilst exploiting the package system to share changes.

As is evident from their construction, prefabs are instances of `GameObject`s. As such they may be instantiated via scripts, which leads us to one of the simple tools created for the game – namely `ObjectSpawner`.

## Tools Development

Armed with the full scripting capabilities of Unity it was possible to create a whole host of different game components with ease. The visual debugging methods and real-time preview provided immediate feedback and allowed for fast prototyping.

### Spawn Points

As a generic prefab instantiator, the `ObjectSpawner` script (and associated class) provides a means for spawning any prefab at a desired location and orientation. If the instance is removed a new instance will spawn after a specified time period.

`ObjectSpawner`s are used to position tokens around the map and also at the start line to spawn the players and AI opponents.

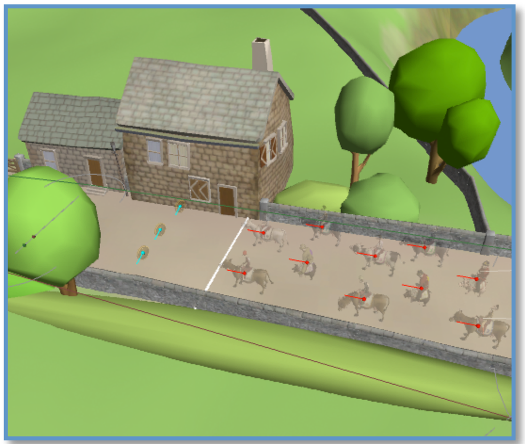


Figure 11 – Spawn points for competitors and boost tokens

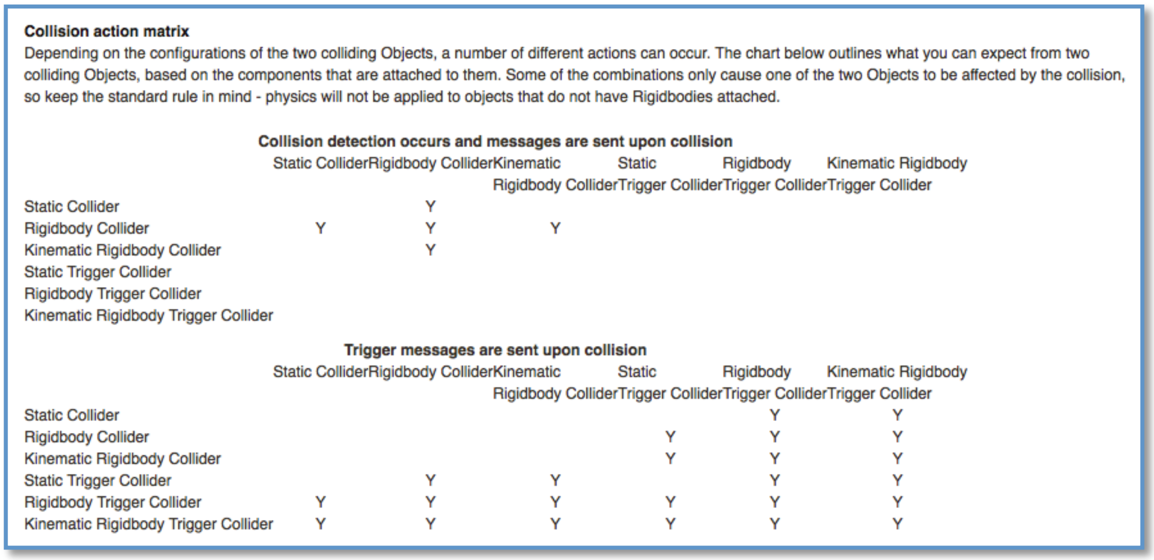
## Checkpoints

Checkpoints allow the game to ensure players are travelling in the correct direction, and that they are not cheating. They also act as a good debugging tool for checking the navigation skills of the AI agents. Missed checkpoints are logged to the console to quickly alert the developer as to issues in the track. If checkpoints are not connected in a loop, a warning is issued.

By measuring the distance between checkpoints and the proportional distance from the previous and next checkpoints to the player, an approximation of the lap position may be estimated. Adding more checkpoints increases the accuracy of this approximation. Comparing the lap position with other competitors provides the player's position in the race.

To make checkpoints detect when a competitor passes one, a collider is attached and it is set to behave as a trigger by enabling its `isTrigger` property. Colliders allow for a variety of physics calculations to be carried out. These calculations are simplified by setting the `isTrigger` flag since no forces need to be applied. More complex physical interactions may be triggered by adding rigid body components, either kinematic or dynamic depending on the requirements.

A full table of interaction between physical entities may be found in the Unity Reference Manual (Unity, 2009 b):



**Collision action matrix**

Depending on the configurations of the two colliding Objects, a number of different actions can occur. The chart below outlines what you can expect from two colliding Objects, based on the components that are attached to them. Some of the combinations only cause one of the two Objects to be affected by the collision, so keep the standard rule in mind - physics will not be applied to objects that do not have Rigidbodies attached.

**Collision detection occurs and messages are sent upon collision**

| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
|---|---|---|---|---|---|---|
| Static Collider | | Y | | | | |
| Rigidbody Collider | Y | Y | Y | | | |
| Kinematic Rigidbody Collider | | Y | | | | |
| Static Trigger Collider | | | | | | |
| Rigidbody Trigger Collider | | | | | | |
| Kinematic Rigidbody Trigger Collider | | | | | | |

**Trigger messages are sent upon collision**

| | Static Collider | Rigidbody Collider | Kinematic Rigidbody Collider | Static Trigger Collider | Rigidbody Trigger Collider | Kinematic Rigidbody Trigger Collider |
|---|---|---|---|---|---|---|
| Static Collider | | | | | Y | Y |
| Rigidbody Collider | | | | Y | Y | Y |
| Kinematic Rigidbody Collider | | | | Y | Y | Y |
| Static Trigger Collider | | Y | Y | | Y | Y |
| Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |
| Kinematic Rigidbody Trigger Collider | Y | Y | Y | Y | Y | Y |

Figure 12 - Collision action matrix in Unity, powered by NVIDIA's PhysX engine

## Constraints

Similar to tools found in many 3D packages constraints allow an object to follow, or copy transformation data from another object. The first iteration of the map screen used such a tool to have tokens follow each character. These were then rendered on top of a simplified model of the track. This idea was replaced by GUI elements since they render much faster and allow for more customisation (see the section Graphical User Interface). However, the simple copy transform constraint paved the way for other, more interesting behaviour scripts to be written.

### Bézier Path Constraint

Using a piecewise Bézier spline, it was possible to build a path and constrain an object to follow this path.

A linear Bézier curve is equivalent to a line, the equation for which is:

$$\underline{p}(t) = (1-t)\underline{p}_0 + t\underline{p}_1$$

Interpolating the two endpoints along two connected line segments provides us with a quadratic Bézier curve. Interpolating the three control points of this curve along three connected segments, with all interpolations using the same t value, gives the cubic Bézier:

$$\begin{aligned}\underline{p}(t) &= (1-t)\underline{p}_0''(t) + t\underline{p}_1''(t)\\ &= (1-t)[(1-t)\underline{p}_0'(t) + t\underline{p}_1'(t)] + t[(1-t)\underline{p}_1'(t) + t\underline{p}_2'(t)]\\ &= (1-t)[(1-t)[(1-t)\underline{p}_0 + t\underline{p}_1] + t[(1-t)\underline{p}_1 + t\underline{p}_2]] + t[(1-t)[(1-t)\underline{p}_1 + t\underline{p}_2] + t[(1-t)\underline{p}_2 + t\underline{p}_3]]\\ &= (1-t)^3\underline{p}_0 + 3t(1-t)^2\underline{p}_1 + 3t^2(1-t)\underline{p}_2 + t^3\underline{p}_3\end{aligned}$$

Resetting the t value for each piece of the spline allows the length to be traversed. The `PathConstraint` script traverses the curve to position an object, traverses it again slightly ahead to gain a forward vector and, using global y as the up vector, aligns the object in the direction of the path.

Each path with constraint is made up of three components:

- `CurvePath` – the object containing information about the path
- `CurvePathNode` – a control point on the path
- `PathConstraint` – the constraining script used to move and orient an object along the path

When a path-constrained object reaches the end of a path, an `OnPathEndReached` event is fired. This allows paths to be looped or for further action to be taken (such as starting the race, in the case of the track preview camera in the game).

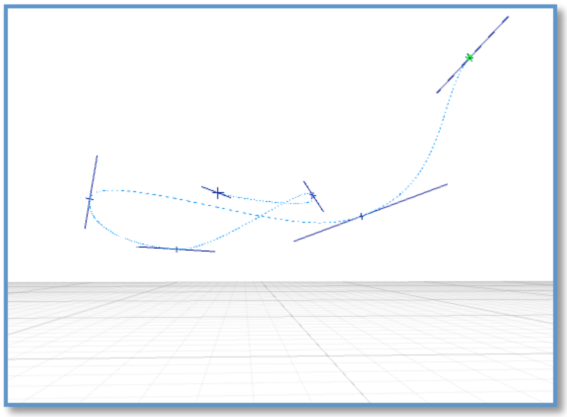


Figure 13 - Bézier path tool

### Collision Mesh Replacement

Each mesh imported for the environment automatically contains a `MeshCollider` component. This requires much more processing than a simple bounding box.

Since the track bounding geometry is relatively simple, a script was written to replace all `MeshCollider`s with `BoxCollider`s recursively from a parent.

The same script (`AddChildColliders`) was later adapted to add other collider (including `MeshCollider`s) wherever they were required.

## Gameplay Programming

### Camera Controller

A dynamic camera was required that would follow the player in a smooth manner. To achieve this the camera was given its own velocity that would accelerate towards its target based on the speed of the target and the distance from it. The camera was also aligned to the target so that the racer was always visible.

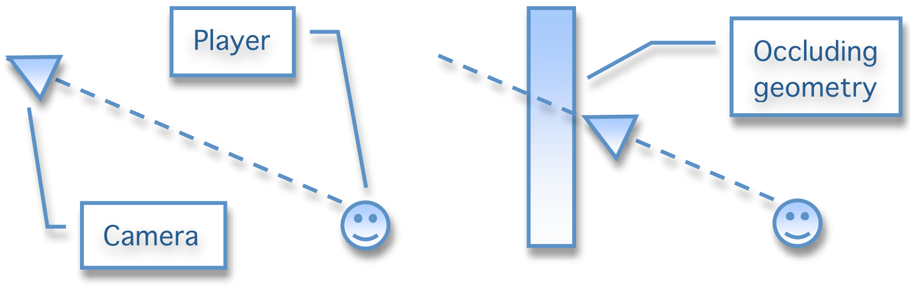


Figure 14 – Avoiding occlusion of the player as viewed from its chasing 3rd person camera

If a ray, drawn from the target to the camera, intersects any geometry, the intersection (offset by the camera radius) is where the camera is positioned. This prevents the camera from becoming stuck outside when the racer enters a building and ensures the player is not occluded.

### Player Controller

One of the more challenging aspects of the project was the behavioural controller for competitors. The `CompetitorController` class forms the largest behaviour script in the project. Not only does it move the player based on input from the player, it also allows control by AI agents through a shared interface. The controller handles geometry collisions and ground interaction, boosting and all game statistics (such as air time, lap times, etc.).

As a basis for non-physical motion Unity provides a `CharacterController` component. This component is ideal for most game characters but has the unfortunate limitation that the collision model is fixed to use a capsule collider. Worse than that, the collider it uses cannot be oriented and as such only vertically standing models can be reasonably contained within its bounds.

The Pig/Policeman character suits the upright container quite well but the cow is much longer than it is tall presenting a difficult problem. By increasing the radius of the controller, the character is no longer able to move close to walls and unrealistically collides with geometry to its sides that are not touching its surface.

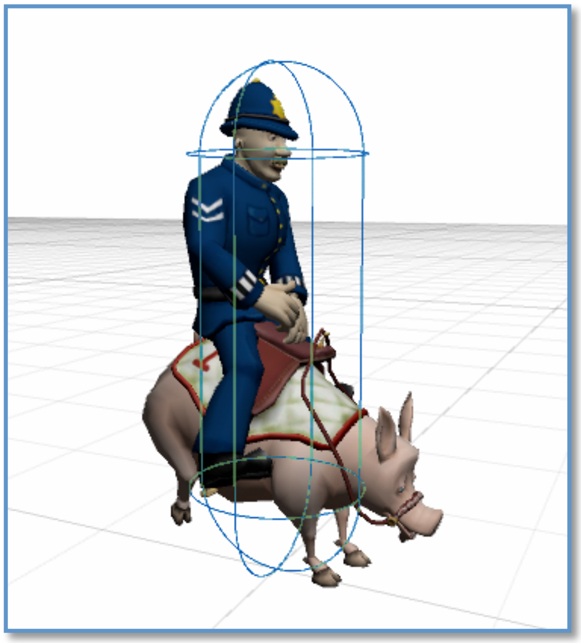


Figure 15 - Character capsule collider

Since the racer is viewed primarily from behind it was important that the silhouette at the least should closely match the collision model in place. It was therefore decided to sacrifice the chance of intersecting geometry for accurate side-to-side collisions.

## Boost Mechanism

From the initial design, the boost mechanism was intended to provide a stepped boost bar that would fill up as tokens were collected. For each section of the bar a different multiplier was to be applied to the time or speed of the boost.

It was decided have the boost empty whatever section was currently being filled so that the player would not be able to stop boosting until the section ran dry. This proved to be unexpected and confused some players (see Testing). The drive behind this decision was that it would make the game more fun, trying to avoid crashing while moving at a fast pace through the level.

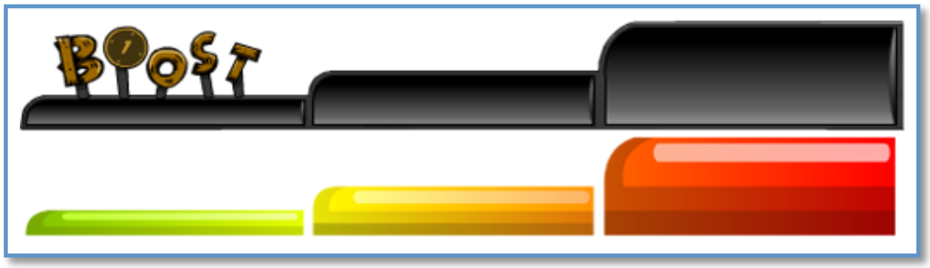


Figure 16 - Boost bar graphics

To avoid this confusion the decision was made to require the holding of the boost button to maintain the boost. This not only made the game less entertaining but also made it very easy to avoid crashing. A compromise was instated, having a minimum boost period where the player could not stop but that to continue boosting the button would have to be held down.

## Token Pick-up

Spawned by an `ObjectSpawner`, boost tokens could easily be placed around the track. Similarly to checkpoints, tokens use trigger enabled colliders to detect the player. Once collected an event is triggered on the colliding racer, and if the boost level is still within the bar it is increased and the token destroyed.

By destroying the token, the `ObjectSpawner` parent is triggered to begin counting down until it is due to spawn another token. Thus tokens respawn.

### Track Boundary Checking

To ensure the player remained on the track it was first decided to block them in by raising the trackside wall colliders. Causing other issues and presenting an invisible wall that confused players, an alternative was sought.

The players (and AI competitors) are now reset to the nearest track node if they venture out of bounds. The checking mechanism uses a low-resolution black and white texture as a mask for world coordinates that are in and out of bounds. This is easily adjusted and has proven to be a successful solution. Players found to be out of bounds are presented with a fading screen and are shrunk out and grown back onto the track.

### Competitor Instantiation

By constructing a prefab from which competitors could be built, the spawning process could be made to be a reasonably automated process. The `Competitor` class first ensures that all other scripts are properly attached, registers itself to gain its race number and then proceeds to use this information, along with the global game settings, to construct a character from the mount and rider prefabs piped in.

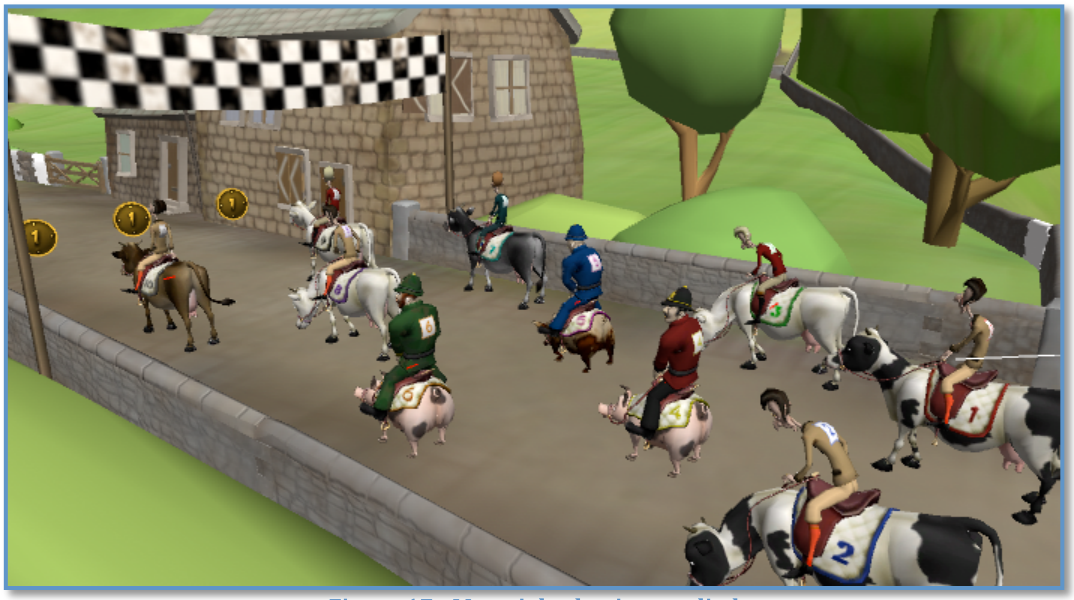


**Figure 17 - Material selection applied**

If the race number is less than the game settings player count then the racer is assigned the rider and mount chosen in the character selection menu with the chosen skin. Otherwise, it is assigned a random rider and mount, and a random skin. (It is thus also a computer-controlled competitor.)

The skin is selected by a `MaterialSelector` script, which acts to set both the skin and race number texture for the character. The race number is assigned to a separate UV layer to allow this integration. Each character therefore has two UV layers: one for the skin and another for the race number (or race numbered tack). The target material indices can be set in each `MaterialSelector` so that it assigns the texture to the correct UV layer.

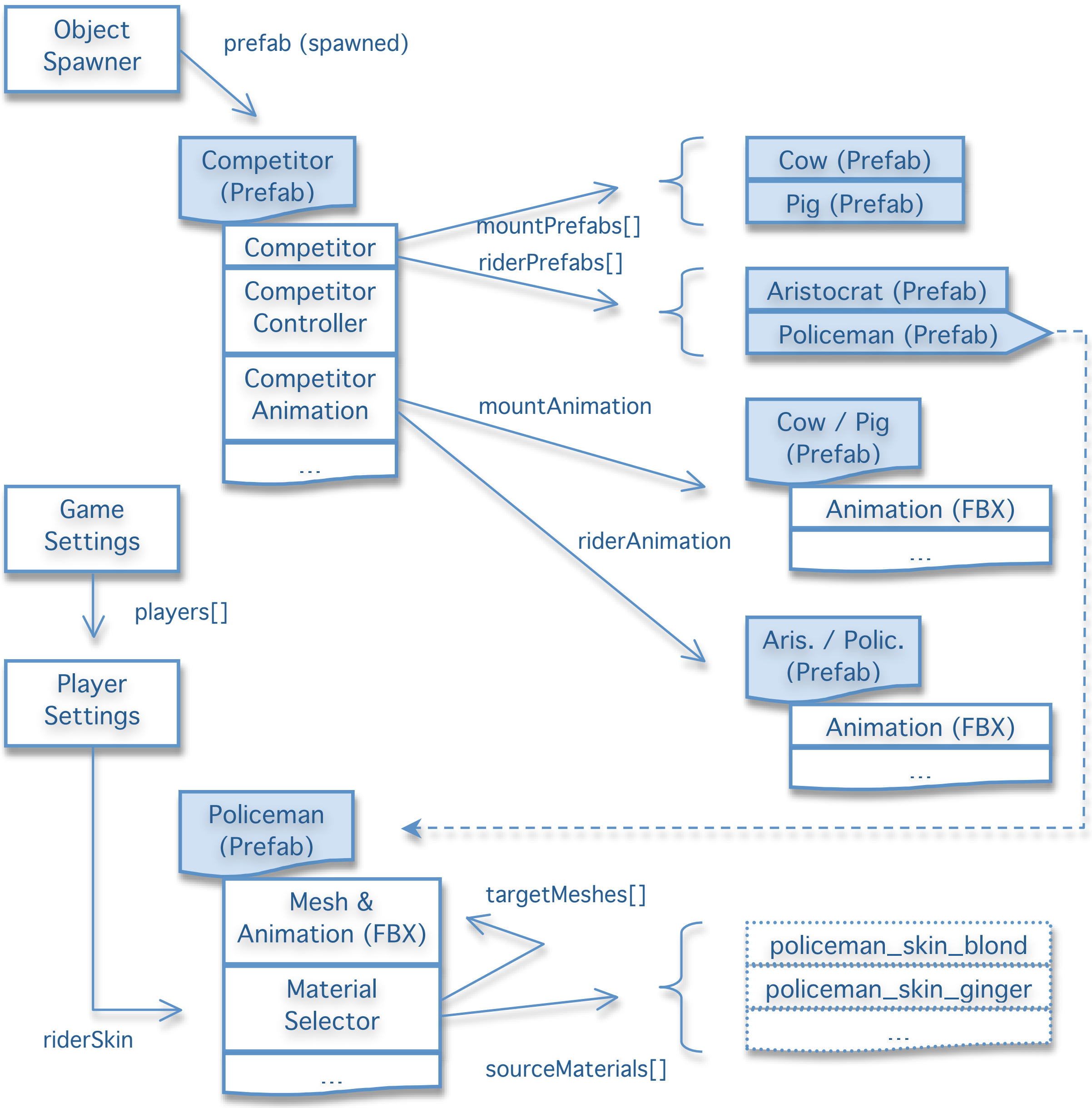


Figure 18 – Spawning a competitor

## Animation

Animation has been key to the success of the game's appeal. Without animation, as was the case in the first few weeks, the game is lifeless. It was decided to have several animations for different animal gaits and to blend between them as the animal accelerates. The rider has a set of animations to match these gaits and extra ones for attacking other players.

### Blending

Based on the speed of the mount, a gait is chosen and speed-matched so that each footfall lands correctly. As the speed of the mount changes a new gait must be blended in. To achieve this an animation stack is used.

Whenever the animation is changed this new animation is pushed to the stack with a weighting of zero, in an animation layer above the previous stack entry. Over time, each stacked animation is increased in weight until it reaches a weight of one. At this

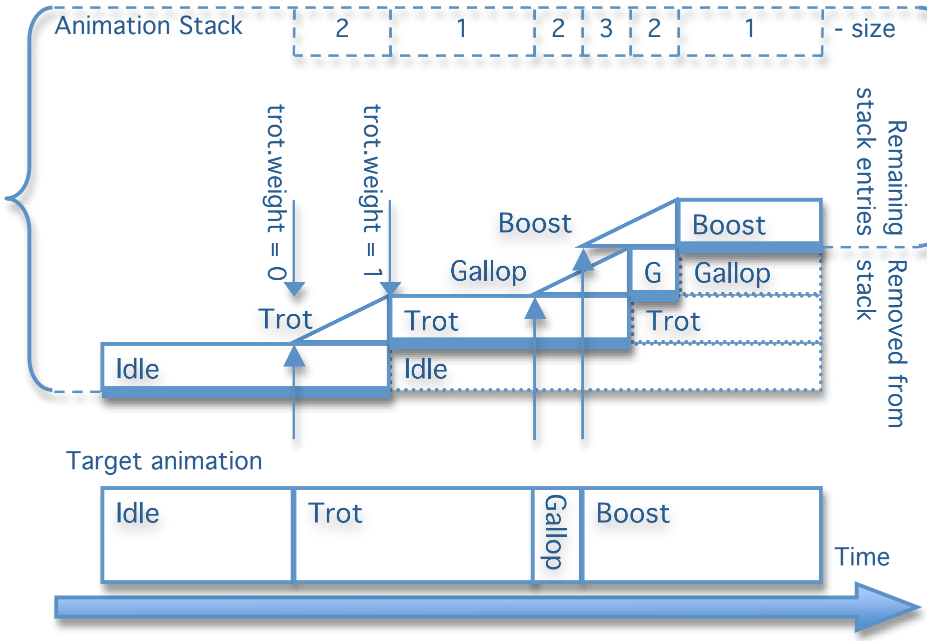


Figure 19 – Animation blending stack

point the stack is partially collapsed: all animations under the fully blended animation no longer affect the output and may be removed. The layer indices for all other animations are also reduced such that the highest layer index is equal to the stack size.

Unity's animation system blends all enabled animations, even those with a zero weighting, so it was important to disable clips upon their removal.

### Speed Adjusting & Limiting

To ensure the acceleration and general motion of each character remained true to the animal's real-life counterpart, certain restrictions were put in place. Each animal had speed limits for each gait, so when accelerating the racer could only go so fast before "changing up a gear". Another imposed restriction was that the animal must travel for a certain time in its trot gait before it will change up to the gallop. Boost was the exception, allowing the player to boost from any state. This was to provide a much more exhilarating experience.

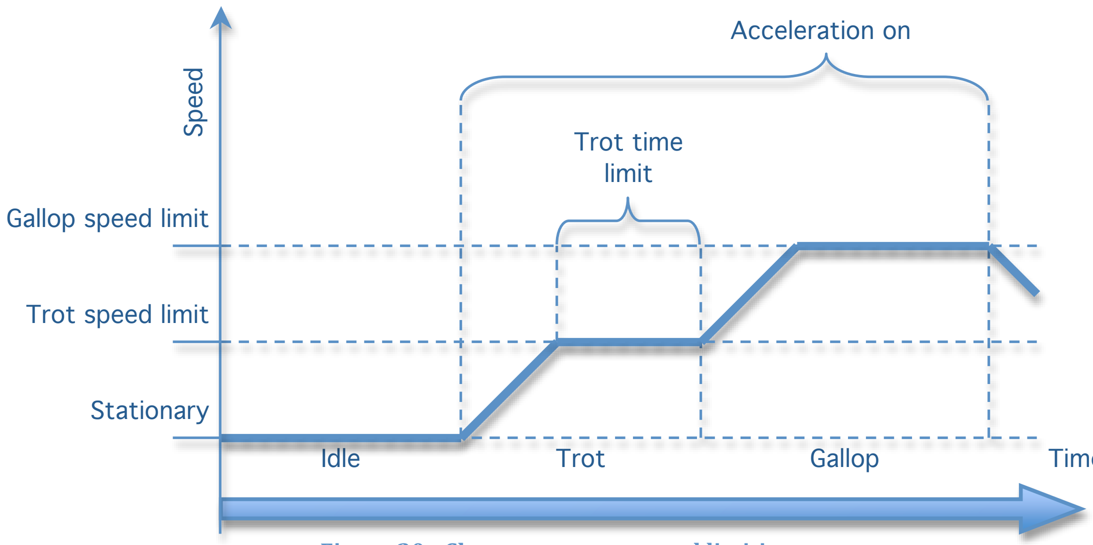


Figure 20 - Character mount speed limiting

### Synchronisation

The rider and mount were designed to be independent so that characters could be mixed and matched. Due to time constraints the animations were not created to cross the Aristocrat and Policeman over. However the structures remain in place for this extension should it be desired. The mounts and riders are rigged and animated independently and as such must be synchronised. Each clip was made to be the same length and the mount and rider animation (excluding attack) were set up to be triggered and blended during the same method call. The attack animation is independent and as such can be layered on top with no adverse effects.

## Artificially Intelligent Opponents

Opponents that could compete with the player were essential for single player mode. For multiplayer mode they also added a great deal to the gameplay. A primitive, artificially intelligent behaviour script drives these competitors to navigate their way around a node-based network.

### AI Network Nodes

Throughout the track nodes are positioned that point the direction to the next node (or a choice of possible "next" nodes). Each AI agent will navigate through this network to complete a lap of the track. There are various choices an agent must make whilst travelling around the track. Some of these decisions, such as the choice of next node when there is an option, are purely random. Other decisions, such as the speed to travel, the position to take on the track and when to boost, are influenced by hints at each node.

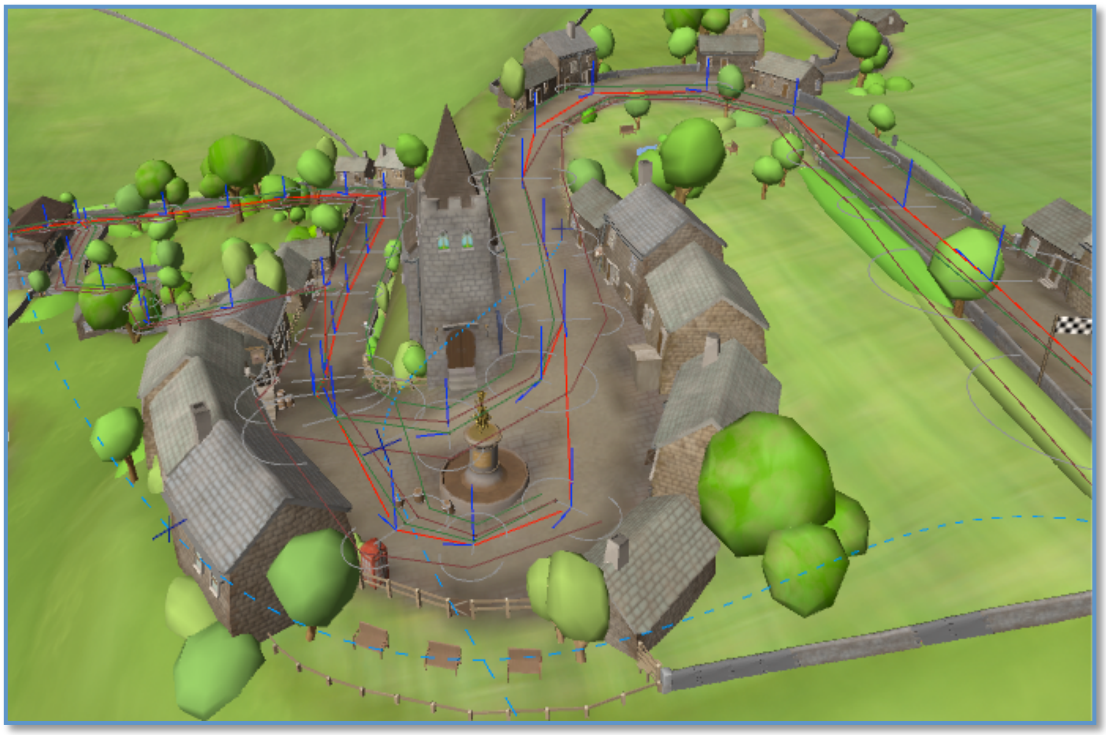


Figure 21 - AI node network

Agents are spread out on the track by a drift parameter. Combined with each node's spread value (providing the limits of drift), a target relative to the next node is ascertained. Trying to reach this point the agent will turn and accelerate as necessary. If the target point is in sight (not occluded in a ray cast), then boosting becomes an option. Boosting is controlled by the speed hint, which provides a probability that the racer should decide to boost between the current and next node.

The node's radius controls the area that must be "hit" before a subsequent node is chosen. While the competitor is within this area it blends towards the next node providing a smoother transition than simply switching target from node to node.

As with all competitors, AI agents adhere to crashing and boundary standards being repositioned on the track as necessary. Additionally, if they become stuck in one place for a period of time greater than a specified timeout, they will be reset and an alternative node chosen at which to respawn. Fluidity is thus maintained.

## Human Interface

This game was built around the concept of using physical Wii remote controllers to steer and otherwise interact with their character. To recover from a crash the Wii remote's accelerometers are exploited to detect shaking. Tilting the remote controls the steering and the buttons provide acceleration, breaking, boost and attack functionality.

Since the game was developed on a PC, keyboard alternatives were also put in place but these are not designed to be primary controls and as such are not as ergonomically laid out.

Pointing the Wii remote at a sensor bar during any menu screen allows point-and-click operation with the D-pad providing alternative input. A physical sensor bar was also built in support of this project, from four IR LEDs, some black card, wires and a battery holder.

### Wii Remotes

Access to the Wii remotes was provided via a Bluetooth connection and the Unity plugin, UniWii (see Unity Overview). This plugin provides functions for obtaining data from any connected remotes but does have some limitations.

Firstly, the data had to be collected, so a `WiiPoller` script and associated `GameObject` (of the same name) were created. Calculations for cursor position, shake count and controller buttons are all handled by this class. If no Wii remote is connected for a particular player, the mouse is polled instead.

Data from the Wii remote accelerometers arrives as an integer value between 0 and 255. To gain a steering value, the force along the length of the remote is measured, the midpoint (near 128) is taken away and the result scaled based on calibration results:

```
// Tweaked values to read zero when flat on a table and
// +1 or -1 when standing on one end
x = -5.2f*(wd.accY-133) / 128.0f;
```

Shaking the remote is detected by waiting for a large positive force in the vertical axis, and then a corresponding large negative force. Each of these forces increments the shake counter, which is read and reset during recovery.

## Graphical User Interface

Unity comes with a set of standard UI components, such as buttons, labels, images, etcetera. However, none of these support multiple cursors and there is no function to position the existing mouse cursor or to simulate a click. It was therefore not possible to use the Wii remotes' point-and-click to "click" anything in the existing library.

Writing a new UI library – which makes use of the existing GUI drawing functions – it was possible to support all that was required to enable Wii remote interaction.

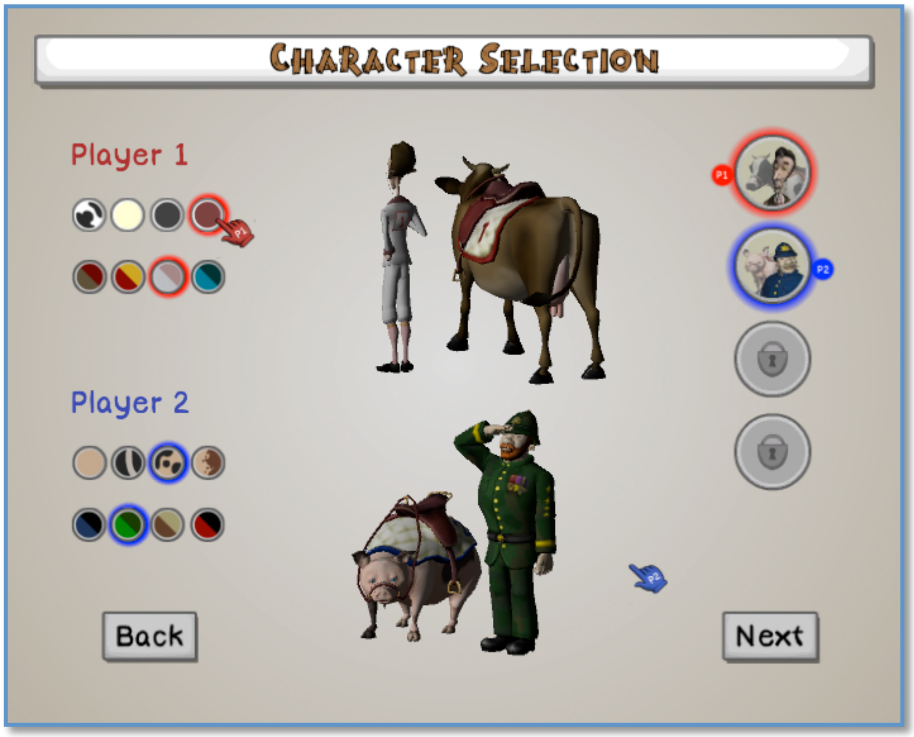


Figure 22 - Character selection UI with multiple cursors

## Design Alterations

The boost system was changed to require holding the boost button, with a short minimum boost time to prevent animations blending too quickly.

Character changeover – that is, the aristocrat on the pig and policeman on the cow – although supported in much of the code was abandoned due to the amount of animation required for the existing combinations.

A jump function was originally meant to be in the game but this was deemed risky when trying to contain the player and as such never made it in, despite the change to allow players to leave the bounds of the track before being reset. If boost is triggered at the correct time the player may "jump" off bridges and hillsides, and can gain at least one bonus token in this manner.

Gestures were vastly simplified; only one is now required. There were to be more gestures for recovery, prompting the player for a randomly selected gesture instead of the now present "Shake!" prompt.

The idea of including mini games was abandoned early to keep focus on the quality of the one game with associated menus and racetrack. The final product is therefore complete without being overly complicated.

Special tokens were simplified to just one: the drunk token. Others would have had more global effects (speed limits, global forced boost, etcetera) but were deemed to be unclear in their effects and would again overcomplicate the game.

## Testing

Testing was carried out on a weekly basis to gauge which aspects of the game were most important to players, what issues there were and which parts were most enjoyable.

Feedback from these testing sessions was collated and summarised in an online, collaborative document. This document may be found on the accompanying DVD.

## Conclusion

The project was successful in that a fun and exciting game has been produced. There are plenty of possibilities for expansion (creating new levels, character combinations and collectables) and there are a few areas for improvement. The main issue at present is the need for more optimisation since frame rates are quite low when playing in two-player mode. Otherwise, the experience has been thoroughly rewarding and at the end of the project a playable and functioning game has been created, with interesting characters and a well-built environment.

## Credits

### Core Team:

Character modelling, texturing, animation and user interface – Sophie Shaw
Environment modelling, texturing, level design and set dressing – John Griffiths
Programming, rigging, technical direction and asset management – Ian Thompson

### Audio:

Sound design – Matthew Kennedy
Music composition – Kirstie Hewlett

### Testers:

Lead tester – Nicholas Hampshire, Michael Cashmore, Peter Agg
Tester – Andrea Miller, Ashley Morrison, Brian O. W., Chris McLaughlin, David Schott, Finella Fan, Holly Potter, James Lewis-Cheetham, James Roberts, Jamie Wood, Jeremy, Jessica Ott, Kirstie Hewlett, Leah Hullinger, Lucy Pike, Martin Lane, Matt Northam, Miriam Bray, Praveen Kumar, Richie Xu Xing, Robin Chater, Susan Sloan, Tom Lewis-Cheetham, Vanessa Salas Castillo, Will Goldstone, Yolande Clerke

Nintendo® and Wii® are registered trademarks of Nintendo Inc. USA

## Figures

Figure 1 – Inspector in Unity ........ 5
Figure 2 - Unitron script editor ........ 6
Figure 3 – Singularity classes ........ 6
Figure 4 - Disappearing limbs when rotation limits are put in place ........ 8
Figure 5 - Tail curl controls ........ 9
Figure 6 - Pig rig with spine twist ........ 9
Figure 7 – Rig synoptic ........ 10
Figure 8 – XSI animation pipeline, first stage ........ 10
Figure 9 – Rig reduction process ........ 11
Figure 10 – Animation clips on a single timeline ........ 12
Figure 11 – Spawn points for competitors and boost tokens ........ 13
Figure 12 - Collision action matrix in Unity, powered by NVIDIA's PhysX engine ........ 14
Figure 13 - Bézier path tool ........ 15
Figure 14 – Avoiding occlusion of the player as viewed from its 3rd person camera 16
Figure 15 - Character capsule collider ........ 17
Figure 16 - Boost bar graphics ........ 17
Figure 17 - Material selection applied ........ 18
Figure 18 – Spawning a competitor ........ 19
Figure 19 – Animation blending stack ........ 20
Figure 20 - Character mount speed limiting ........ 21
Figure 21 - AI node network ........ 22
Figure 22 - Character selection UI with multiple cursors ........ 24

## References

### Software

Unity Technologies. 2009. *Unity* (2.5) [computer program]. Frederiksberg, Denmark: Unity Technologies.

Softimage. 2008. *Softimage XSI* (7.01) [computer program]. Montreal, Quebec, Canada: Softimage.

Adobe systems. 2007. *Adobe Photoshop* (CS3) [computer program] California, US: Adobe Systems.

### Computer Games

Monster Games. 2007. *Excite Truck* [computer game]. Kyoto, Japan: Nintendo.

Nintendo EAD. 2008. *Mario Kart Wii* [computer game]. Kyoto, Japan: Nintendo.

### Articles

Tulip, J., Bekkema, J., and Nesbitt, K. 2006. Multi-threaded game engine design. In *Proceedings of the 3rd Australasian Conference on interactive Entertainment* (Perth, Australia, December 04 - 06, 2006). ACM International Conference Proceeding Series, vol. 207. Murdoch University, Murdoch University, Australia, 9-14.

### Unpublished

Griffiths J. R., Shaw S. K., Thompson I. P., 2009. *Hamlington Underground Racing Association*. CAPT Assignment Production Diary, (MA / MSc), Bournemouth University.

### Websites

Novell, 2009. *Main Page – Mono*. Available from:
http://www.mono-project.com/Main_Page [Accessed August 2009]

Unity, 2008. *Unity 1.0.1*. Available from:
http://www.gamesindustry.biz/articles/unity-1-0-1-3d-dev-tool-launched-for-the-iphone-and-ipod-touch-platform [Accessed August 2009]

Unity, 2009a. *UNITY: Game Development Tool*. Available from:
http://unity3d.com [Accessed August 2009]

Unity., 2009b. *Unity Scripting Reference*. Available from:
http://unity3d.com/support/documentation/Components/index.html
[Accessed August 2009]

## Appendices

### Appendix A: Class Inheritance

- AddChildColliders
- AIAgent
- AIPathNode
- AnimationTest
- AveragePositionController
- ButtonEvent
- CameraController
- CharacterMenu
- CharacterSet
- Checkpoint
- CollisionTester
- ColorList
- Competitor
- CompetitorAnimation
- CompetitorController
- ConstraintController
- CurvePath
- CurvePathNode
- GameSettings
- GameSounds
- GameUI
- MainGame
- MainMenu
- MaterialSelector
- MeshMerger
- NullMarker
- ObjectSpawner
- OptionsMenu
- OutOfBoundsTrigger
- PathConstraint
  - PreviewCameraController
- GameSettings.PlayerSettings
- Profiler.ProfilePoint
- Profiler
- QuickTest
- RaceParameters
- RaceStatistics
- RiderPrefab
- SimpleAnimation
- SoundSampler
- SoundSource
- Token
  - BeerToken
  - BoostToken
- UIBase
  - HUD
  - MapUI
  - ScoreboardUI
  - StatusUI
  - UIBackdrop
  - UIButton
    - UICroppedButton
    - UISkinButtons
  - UIButtonBlock
  - UICredits
  - UICursors
  - UIImage
    - UICroppedImage
    - UISplashScreen
  - UILabel
- UIManager
  - MenuSystem
  - PauseMenu
- ValidityChecks
- Water
- WiiPoller.WiimoteData
- WiiPoller