

3D VIDEO GAME CREATION IN C#

by

JESSIE SLAMKA

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE HONOURS

in

The Irving K. Barber School of Arts and Sciences

(Honours Computer Science Major Computer Science)

The University of British Columbia

(Okanagan)

April 2011

© Jessie Slamka, 2011

ABSTRACT

3D game development requires the use of various tools for the different stages, from conception, design, modelling, animating, coding, and testing. The focus of this project was to learn how to code in C# specifically for game development with freeware programs and class libraries. The three tools that were explored in the most detail were Blender, Microsoft's XNA in conjunction with Visual C# 2008 Express, and Unity, while touching briefly on Ox and a couple other physics engines that are offered under GNU licences as well as a couple image editors. Challenges primarily lay in figuring out how to adapt previous knowledge of similar programs to the new software and in simulating the physics on the player-controlled character, particularly in terms of projecting motion onto trees.

TABLE OF CONTENTS

ABSTRACT	2
ILLUSTRATION INDEX	4
1 INTRODUCTION	6
1.1 MOTIVATION	6
1.2 GOALS AND OBJECTIVES	7
1.3 CONTRIBUTIONS	8
2 BACKGROUND	9
2.1 TERMS	9
2.2 BLENDER	10
2.2.1 <i>Modelling Walkthrough</i>	16
2.2.2 <i>Animating Walkthrough</i>	19
2.2.3 <i>Applying Materials Walkthrough</i>	20
2.3 XNA IN VISUAL C# 2008 EXPRESS	23
2.4 UNITY	24
2.5 VISUAL STUDIO 2010	25
2.6 GIMP	26
3 CHAMELEON GAME	27
3.1 CONCEPTION	27
3.2 DESIGN	27
3.2.1 <i>Chameleon</i>	27
3.2.2 <i>Snake</i>	29
3.2.3 <i>Environment</i>	29
3.3 MODELLING	29
3.3.1 <i>Chameleon Modelling</i>	29
3.3.2 <i>Snake Modelling</i>	34
3.3.3 <i>Tree Modelling</i>	39
3.4 ANIMATING	43
3.4.1 <i>Chameleon</i>	43
3.4.2 <i>Snake</i>	43
3.5 SCRIPTING	43
3.5.1 <i>Chameleon</i>	43
3.5.2 <i>Snake</i>	45
3.5.3 <i>Environment</i>	45
3.6 TESTING	45
3.7 SUMMARY OF TASKS	46
4 GAMEPLAY	48
5 CONCLUSIONS	49
6 FUTURE WORK	50
7 REFERENCES	50

ILLUSTRATION INDEX

FIGURE 1: SCREENSHOT FROM BATTLEZONE FROM ATARI (1980)	6
FIGURE 2: SCREENSHOT FROM I, ROBOT FROM ATARI (1983)	6
FIGURE 3: MODEL VIEW (CUSTOMIZED FROM FACTORY DEFAULT)	11
FIGURE 4: BLENDER SET UP FOR RIGGING (MODEL LAYOUT).....	12
FIGURE 5: WEIGHT PAINTING MODE WITH PAINTING MASK ENABLED.....	12
FIGURE 6: THE ANIMATION LAYOUT.	13
FIGURE 7: IPO CURVE EDITING SCREEN.	13
FIGURE 8: THE ANIMATION LAYOUT.	14
FIGURE 9: THE MATERIAL LAYOUT.	15
FIGURE 10: AN EXAMPLE OF A LOGIC BLOCK SETUP	16
FIGURE 11: INITIAL CUBE WITH TOP FACE SELECTED	16
FIGURE 12: TOP FACE EXTRUDED AND ROTATED	17
FIGURE 13: MANIPULATING A VERTEX.....	17
FIGURE 14: APPLYING LOOP SUBDIVIDE TO MESH.....	17
FIGURE 15: MESH AFTER LOOP SUBDIVIDE APPLIED.	18
FIGURE 16: PULLING OUT VERTEX ON THE EDGE CREATED BY LOOP SUBDIVIDE.	18
FIGURE 17: MESH WITH SUBSURF APPLIED.....	18
FIGURE 18: SELECTING POSE MODE IN A 3D VIEWPORT CONTROL PANEL	19
FIGURE 19: TRANSLATING A BONE POSITION	19
FIGURE 20: ACTION EDITOR SCREEN	19
FIGURE 21: THE PREVIEW WINDOW	20
FIGURE 22: THE MATERIAL, PREVIEW, SHADER, AND MIRROR TRANSPARENCY BUTTON GROUPS,	20
FIGURE 23: MAP INPUT BUTTONS GROUP	21
FIGURE 24: MAP TO BUTTONS GROUP.....	21
FIGURE 25: AN ALTERED MARBLE TEXTURE.	21
FIGURE 26: TEXTURE TYPE OPTIONS	21
FIGURE 27: TEXTURE APPLIED TO ALPHA (TRANSPARENCY) CHANNEL.....	22
FIGURE 29: UNITY WITH VIEWPORT (TOP LEFT), GAMEPLAY SCREEN (BOTTOM LEFT), HIERARCHY VIEWER (MIDDLE TOP), PROJECT VIEWER (MIDDLE BOTTOM), AND INSPECTOR WINDOW.....	24
FIGURE 30: VISUAL STUDIO 2010 WITH SIDE-BY-SIDE EDITORS RUNNING. TO THE LEFT IS THE SOLUTION EXPLORER, WHICH DISPLAYS ALL PROJECT FILES AND FOLDERS. CURRENTLY, A NATIVE JAVASCRIPT FILE IS BEING TRANSLATED INTO C#.	25
FIGURE 31: CREATING AN ALPHA MAP IN GIMP.....	26
FIGURE 32: SIDE VIEW OF YOUNG VEILED CHAMELEON USED AS REFERENCE IMAGE DURING MODELLING	28
FIGURE 32: BACKGROUND REFERENCE ADDED IN SIDE VIEW. BUILT MESH ACCORDING TO IT.	30
FIGURE 33: LEG IN PROGRESS. ARM MOSTLY FINISHED	30
FIGURE 34: SUBSURF MODIFIER APPLIED TO SMOOTH THE LOW RESOLUTION MESH.	31
FIGURE 35: ADDING CUTS TO THE LEG (PINK - IN PROGRESS, YELLOW - JUST ADDED).	31
FIGURE 36: ADJUSTING WHERE THE CUT WILL OCCUR ON THE LEG ALONG THE GREEN EDGE.....	31
FIGURE 37: THINNING THE TAIL RELATIVE TO THE WIDTH OF THE BODY.....	32
FIGURE 38: ADJUSTING THE WIDTH OF THE TORSO.....	32

FIGURE 39: CUTTING THE EYE SO IT WILL BE ABLE TO HAVE THE CORRECT "DOME" SHAPE. PURPLE – CUTS IN PROGRESS.33

FIGURE 40: NOTE THE GAPS IN THE COLOURS (PAINTED WEIGHTS). USING THE PAINTING MASK TOOL (RIGHT) HELPS HIGHLIGHT MISSED VERTICES VERSUS USING THE NORMAL WEIGHT PAINT VIEW (LEFT).33

FIGURE 41: FIXED TAIL. IT CURVES NICELY NOW.34

FIGURE 42: FINISHED SNAKE MODEL VIEWED IN POLYGON EDIT MODE.....34

FIGURE 43: FINISHED SNAKE MODEL WITH SUBSURF MODIFIER APPLIED.35

FIGURE 44: CREATING ARMATURE BONES. THE BONES THAT WILL BE DEFORMING THE SNAKE ARE THE ONES LYING AT THE BOTTOM OF THE SNAKE MESH.35

FIGURE 45: ADDING THE FIRST CONTROL BONE (YELLOW) TO THE SPINE BONE CHAIN.....36

FIGURE 46: ALL CONTROL BONES IN PLACE ALONG TOP ROW.36

FIGURE 47: COMPLETED CONSTRAINTS: YELLOW IS IK SOLVER (SPINE BONE POINTING AT ITS CORRESPONDING IK BONE), GREEN IS STRETCH TO, WHICH POINTS AT THE NEAREST CP BONE IT POINTS AT (HALF ARE CIRCLES HERE).37

FIGURE 48: JUST CONTROL POINT BONES VISIBLE. MESH IS COMPLETELY RIGGED.38

FIGURE 49: RIGGED WITH SPINE HAVING B-BONES WITH THREE SEGMENTS EACH39

FIGURE 51: FINISHED TREE WITHOUT SUBSURF MODIFIER.40

FIGURE 52: FINISHED TREE WITH SUBSURF MODIFIER APPLIED. NOTE THE PLANES ON THE BRANCH (PINK – SELECTED). THESE HAVE A LEAFY TWIG IMAGE APPLIED TO THEM.41

FIGURE 53: FINISHED TREE PREVIEWED WITHOUT LIGHTS.42

FIGURE 54: TEXTURE TO SIMULATE LEAFY TWIGS.....42

1 INTRODUCTION

Video games had their start in two dimensions in the late 1940s with the creation of missile defense systems monitored on cathode ray tube displays. The idea of the monitoring programs was later developed into games during the 1950s. The first truly 3D games did not appear on the scene until 1980 with Atari's Battlezone. Its method of display was vector based and included no filled polygons (surfaces). Surfaced 3D graphics did not appear until 1983, with Atari's I, Robot, which is commonly considered the forefather of the modern video game.

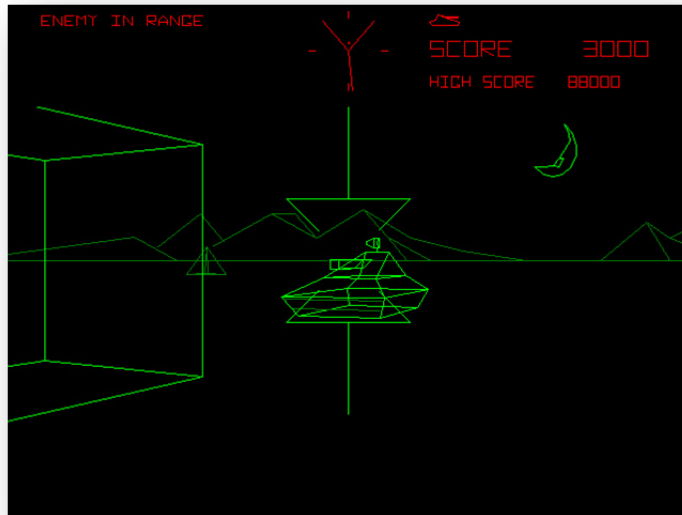


FIGURE 1: SCREENSHOT FROM BATTLEZONE FROM ATARI (1980)

Since then, the complexity of graphics, the detail of game logic, and the effectiveness of player interaction with the game has increased. 3D games have soared in popularity, mostly due to the immersive experience they can provide their players. Unlike 2D games, terrain is generally more unrestricted and the relation of the character's movements to movements the player would make is greater.



FIGURE 2: SCREENSHOT FROM I, ROBOT FROM ATARI (1983)

1.1 MOTIVATION

This project was to investigate what amateurs intent on developing a 3D game have available to them in terms of freeware to perform the game design process. Three primary software requirements exist: a 3D model development suite capable of creating necessary animations for characters, an image editing program that can generate the appropriate textures to colour the models, and a coding suite with a library of classes capable of importing sound, image, and model elements along with their animations. Professionals have many other tools available besides the ones previously mentioned, customized by their in-house software development teams in

some cases. Amateurs are again gaining a foothold in the market, with listings such as Xbox Live Marketplace to sell their games on, and with more widespread possibilities in terms of platforms to develop for, from the traditional PC, to consoles like Xbox, to iPhones and other cell phones.

The secondary motivation was to find out how video game 3D animation differed from the animation done for cinema, advertising, and broadcasting. In film, models are entirely animated in the environment they will be seen in: the characters are animated in their environment and their environment is animated to react to them. In video games, all animations are either prefabricated for specific models (character walk cycle, turn, jump, etc.) or handled by programming or the physics simulator. As the player is controlling the character, animations must be flexible enough to handle anything the player may make the character do within the permitted actions and given game environment.

1.2 GOALS AND OBJECTIVES

As most amateurs have little in the way of funds to spend on their pet projects, the goal was to figure out what tools were available to develop a game for a PC for the least cost. The software solutions investigated were therefore primarily under GNU License and available for download on the internet. Optimally, the tools would be suited to someone with a more visual focus so there was a limited programming requirement.

These pieces of software were evaluated based on their design and how well they assisted in development of a game that would be playable first and foremost on a PC with a current Windows operating system. For the code development suite, it had to support programming in C# and provide a baseline library with vectors and common 3D operations like translation, rotation, and scaling. The code library also had to support importing sound (.wav files), graphics (bitmaps, gifs, and jpegs), and animated models (.fbx files). For the 3D modelling suite, it had to support building and editing models, had to allow these models to be rigged with bones to allow them to be animated more easily, had to support animation, had to export animated models in the .fbx format, had to support the colouring of models, and had to have a minimal number of modifiers, specifically ones geared toward smoothing low resolution models and reducing the resolution of high resolution models. This program in particular had to be highly visual—it had to provide viewports for viewing the model during development. This was important as developing and animating a model is difficult without visual feedback. The final baseline program, the image editor, had to also be highly visual, support selection based on colour values, and be able to output images in a variety of file formats (specifically .jpg, .bmp, and .gif). Support for a tablet was also a requirement, as generating textures by hand with a mouse is difficult.

The goal for the game produced was that it would be re-playable (would be designed such that the player would not mind playing it more than once), have a goal for the player to reach during gameplay, and have obstacles that the player would have to overcome. The goal for the game's models was that they correctly resemble what they are to the highest degree possible while maintaining a model resolution low enough to not impede the computer's ability to render the graphics. The animation of the models would optimally blend well from one action to another and reflect the actual motion of the model's source of inspiration (i.e. a chameleon model's walk cycle would reflect the walk cycle of an actual chameleon).

1.3 CONTRIBUTIONS

The contributions of this thesis include a survey and experimentation with freeware software used for 3D game development in order to ascertain its usefulness and competitiveness in the current market, and a 3D game produced using these software solutions that has a player controlling a chameleon searching for an item in a 3D environment.

2 BACKGROUND

2.1 TERMS

Animating – the process of creating movements for a given mesh, either freehand, manipulating the mesh itself, or through the use of an underlying “skeleton”.

Armature – Blender’s name for the object type that is used as a “skeleton” system for a mesh. Armatures consist of a collection of bone objects which generally exist in a hierarchy of the creator’s design.

Boo – a programming language; Unity supports scripts written in this language.

C# – an object-oriented programming language developed by Microsoft; the language primarily used to code game logic.

Child – in this context, a child is an object that inherits all manipulation made to the parent object; so if the parent is moved three units to the left, the child will mimic that motion from wherever it is.

Constraint – a rule imposed on an object in the context of development in Blender.

Edge – a line connecting two vertices.

FBX – a format developed by Autodesk that allows 3D meshes and their associated animations to be exported to other programs; the format Unity uses to bring in Blender elements; a format that XNA supports for bringing in 3D elements.

Face - a surface bounded by three edges that has a normal indicating the direction from which it can be perceived unless it is set to be double-sided; the number of faces dictates the resolution of the mesh and the level of curvature (if there is any) that is depicted.

Forward Kinematics – the usual constraints applied to armature object bones are used to limit the motion; the parent-child relationships of the bones dictate the way the bones move when the animator manipulates them.

Frame – a unit of animation; essentially one drawing in the sequence of drawings that becomes the animation clip. One standard is 30 frames per second (fps).

Inverse Kinematics – a constraint commonly used during the rigging process to describe the limitations of movement for a chain of bone objects. The “tail” bone of the Inverse Kinematic chain will be the target of motion for all the bones in the chain if no alternate target is specified. The chain of bones goes from the tail bone up the parent-child hierarchy to a specified point, reversing the usual order of manipulation so the child bones influence the motion of their parent bone, thus “inverse motion.” The bones in the chain will move any way possible within their constraints in order to reach or point to their target bone. IK chains are primarily used in limbs so that hands and feet stay on target during an animated motion. This essentially allows an animator to manipulate the bones of the chain backwards.

JavaScript – a programming language; Unity supports scripts written in this language.

Mesh – a collection of vertices, edges connecting them, and faces that exists as a graphical 3D element in a game.

Modelling – the process of creating the desired mesh for a game.

Parent – in a parent-child relationship, the parent imposes all manipulations made to it to its children, including movement, rotation, and changes in scale.

Polygon – similar to a face in a mesh, save that polygons are made up of one or more faces. A square polygon is composed of two triangular faces.

Python – a programming language; used by Blender.

Resolution – the level of detail of a mesh; the number of polygons a mesh is made of. For example, meshes that you find in movies (e.g. Gollum from *Lord of the Rings*) have millions of polygons in order to appear smooth and to create details like the little wrinkles in the skin. A low resolution mesh is a cube: it only has six polygons.

Rigging – the process of adding bones to create a “skeleton” that would produce the desired control over a mesh and “painting” weights on vertices of the mesh that correspond to the desired bone.

Vector – a data construct used by XNA and Unity that contains dimensional information – X, Y, and Z when using a 3D vector.

Vertex – a point in 3D space—has an X, Y, and Z coordinate.

Weight Painting – the process of “painting” influence for a given bone on vertices of a mesh; for example, all the vertices in the arm of a mesh would be painted with an influence of one for the arm bone and left with an influence of zero for the head bone, this would force the arm vertices to move as the arm bone does and not react at all when the head bone moves.

XNA – a library of C# classes released by Microsoft to act as a basis for programming games compatible with Windows operating systems and Xbox consoles.

2.2 BLENDER

Blender is a 3D studio program that includes a game engine capable of running on most operating systems. It is currently at a beta release of version 2.5. Version 2.49b was used as the beta was not compatible with Unity. Tutorials and manuals are available online, and there are a number of forums dedicated to this software solution. It is one of the leading freeware competitors to market solutions such as Maya (Autodesk), 3D Studio Max (Autodesk), Softimage (Autodesk), Animation Master (Hosh), Aladdin 4D (DiscreetFX), and many others.

Modelling and animating game elements requires an application capable of generating 3D meshes, texturing them, and manipulating them at a higher level with a pseudo-skeleton. Autodesk has dominated the playing field of what is widely available with XSI, Maya, and 3DS Max. Most companies like EA and Ubisoft either use these products or have their software teams develop their own. These programs are on the range of thousands of dollars for a professional copy. These prices are justified, given the amount of work that has gone into all the features in these programs, from their physics engines to their design tools, but they are beyond a university student’s reach.

Notably, it is very difficult to learn to use Blender if the user previously used some other comparable software. Switching from ingrained 3D Studio Max commands to Blender hotkeys was difficult. However, after progressing significantly through the learning curve, modelling and animating workflow is faster as the user is not constantly hunting for buttons to click to switch from rotation to translation or something similar.

It requires little in terms of hardware and disk space. The interface is comprised of a number of screens, which can be switched from displaying a viewport displaying the current meshes being worked on to a buttons interface for mesh editing. The layout of the different screens is highly customizable, as screens can be split, resized, and their contents changed. For game development, most work is in three basic layouts: animation, material, and model.

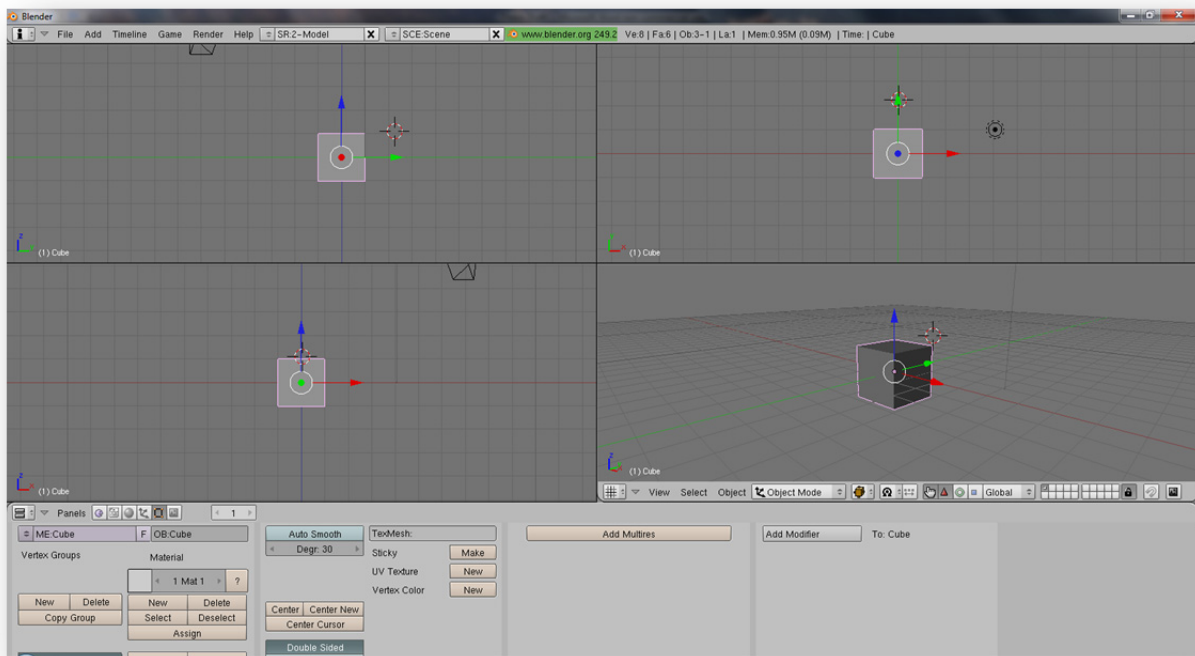


FIGURE 3: MODEL VIEW (CUSTOMIZED FROM FACTORY DEFAULT).

The model layout default can be customized from having one 3D viewport screen to four: right, top, front, and perspective views. Right, top, and front views are orthographic, which means they do not display in terms of a vanishing point, instead drawing everything as an absolute flat projection. Perspective view incorporates the vanishing point, allowing the user to see how Blender will render their objects in proper perspective as they would be perceived by the human eye. These four views can be made to display objects as wireframes, textured, shaded, or bounding boxes. Four is an optimal amount of views for modelling as the mesh is projected in all three dimensions and then can be seen as is in perspective. The bottom part of the screen is a screen that displays the buttons screen, which allows the manipulation of object properties.

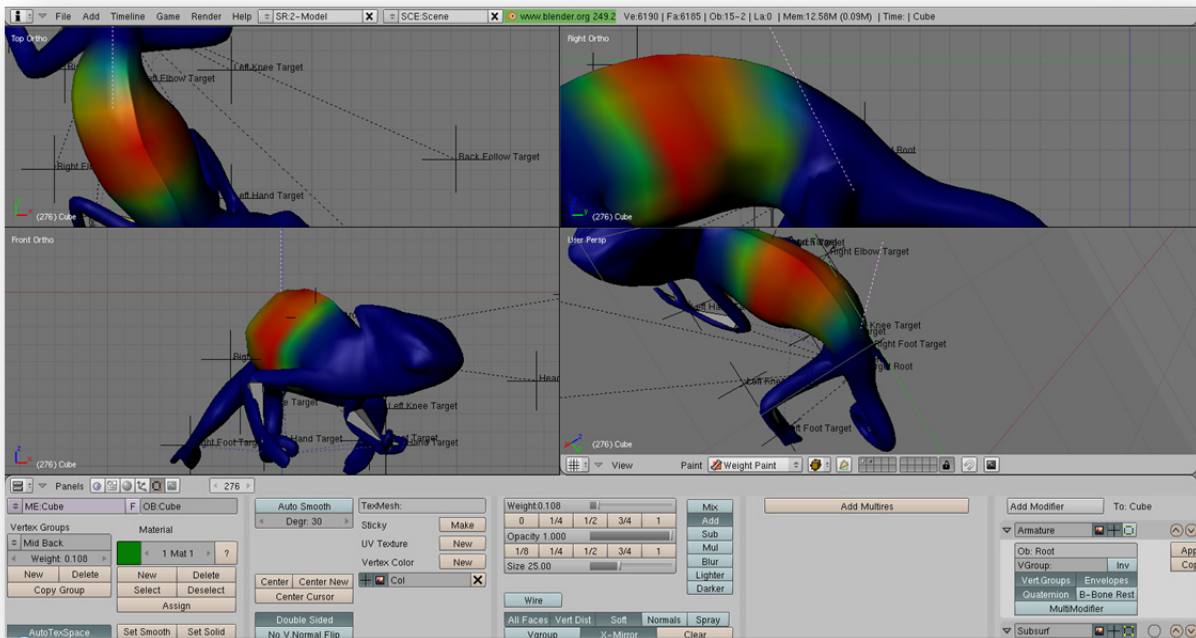


FIGURE 4: BLENDER SET UP FOR RIGGING (MODEL LAYOUT).

Modelling and rigging is best done in this view as these two activities require the most visual involvement with the mesh and any objects involved in the rig. Weight painting, as seen in fig. 4, should be done in the modelling layout as the effect of a bone upon a mesh is displayed through colours, as seen above. The amount of influence a bone has over a mesh's vertex is indicated with colours, red being 100% influenced (1.0) and dark blue being 0% (0.0)

influenced by the bone. The difference between the dark blue of 0.0 and 0.05 can be a little too subtle to perceive. An additional tool, Painting Mask, is available in weight paint mode. It removes shadows from the mesh and causes all colours to become their neon equivalent, allowing subtle differences to be more easily distinguished. It also draws the final mesh as a wireframe overlay, to make it more obvious where the vertices are.

The Painting Mask tool must be disabled in order for the user to select a different bone to weight paint.

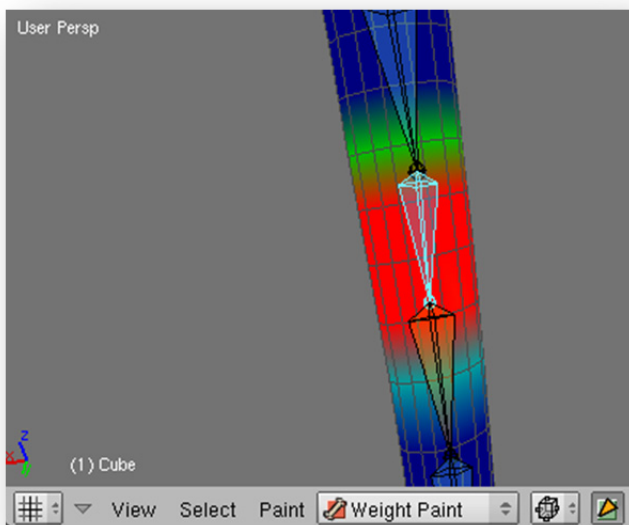


FIGURE 5: WEIGHT PAINTING MODE WITH PAINTING MASK ENABLED

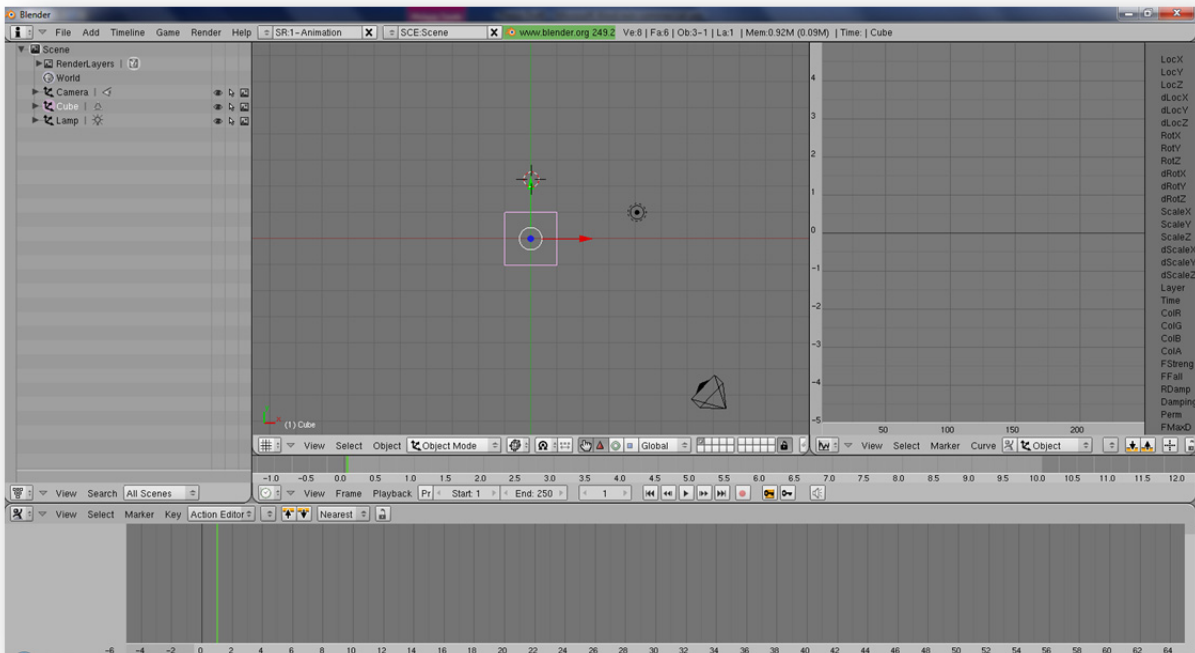


FIGURE 6: THE ANIMATION LAYOUT.

The animation layout out of the box has only one 3D viewport; two views is generally enough to get a good feel for the X,Y, and Z changes without making the view too small, so splitting the viewport worked best, as seen in fig. 8. Configuring these two screens as left and top worked best when working on the chameleon, but left and front would be a better option for bipedal characters. To the left of the viewing area is the hierarchy screen, a list of all the elements in the scene.

To the right of the viewing area is the IPO curve editor screen. This shows the key frames (animation points) the user has generated for the selected object, dividing properties such as location and scale into their X, Y, and Z components, allowing each axis to be manipulated independently. Rotation information is a special case: it is divided into four quaternions instead for some objects. Each

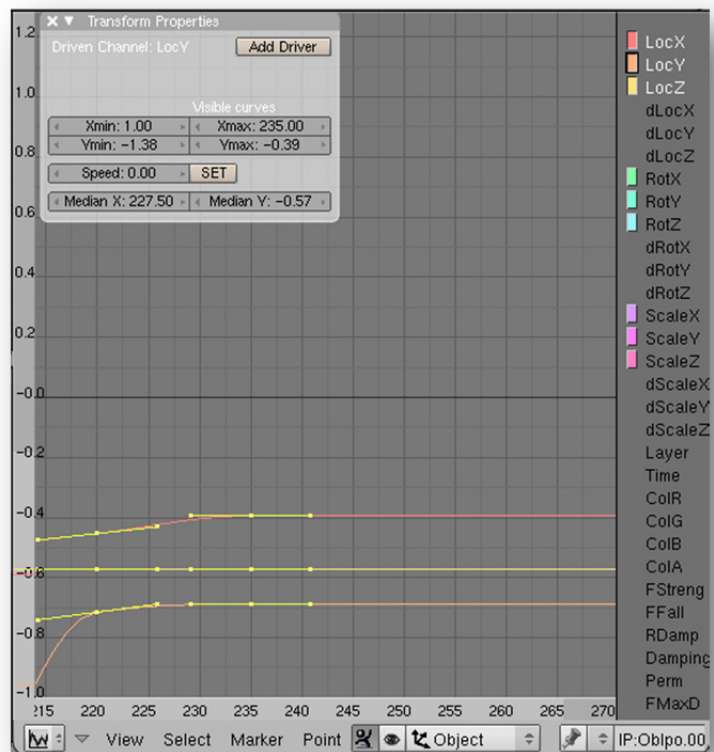


FIGURE 7: IPO CURVE EDITING SCREEN.

property's animation is manifested as a curve and each key frame appears as a point on this curve. Controlling the curve between animation points is done with Bezier handles that can be set to a variety of options.

Note in fig. 7 how rotation, scale, and location are all assigned colour blocks (right-hand side). This indicates that there are key frames for those properties. Note how location blocks have white text, which means that they have been selected to be displayed in the curve editing screen. Location blocks correspond to the curves displayed in the matching colour. There are two key frames visible in fig. 7: one at frame 220 (LocX is approximately -0.45, LocY -0.71, and LocZ -0.58) and the other at 235. The points to either side of each key frame are the Bezier handles. These are only displayed when the curve is in edit mode. The yellow colour of the point and the handles indicates that they have been selected.

Also visible in fig. 7 is the Transform Properties window, a floating window that can be called up in any 3D viewport and the IP curve screen. It allows the specific properties such as location, rotation, and scale to be viewed and manipulated numerically and locked to further editing if necessary. In the IPO curve editor screen, the Transform Properties displays a specialized set of data: the max and min points of the visible curves, the speed of the animation between points, and the median X and Y. In the case of fig. 7, the median X is 227.50 because the two sets of key frames have been selected—220 and 235—and 227.5 lies at their midpoint. If only one key frame were selected, the median would reflect that key frame's location on the timeline. The median Y reflects how all six points selected center around -0.59.

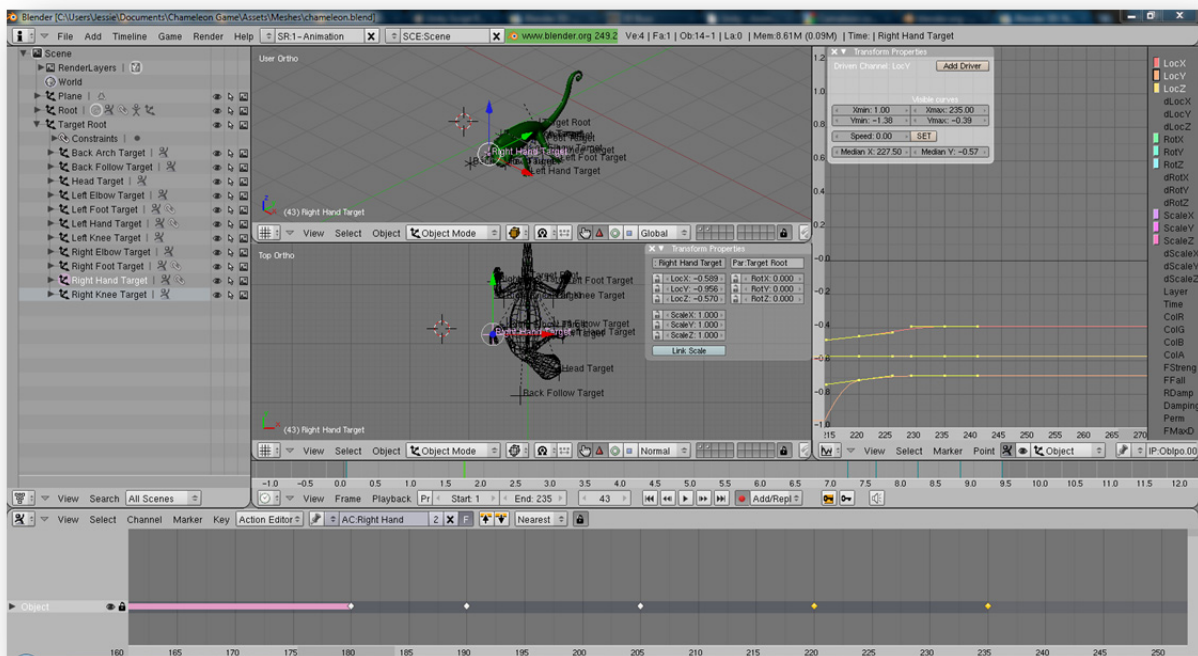


FIGURE 8: THE ANIMATION LAYOUT.

Below the viewing area and the IPO curve display area is the timeline section of the screen. It displays the events occurring at specified key frames for the selected object as coloured lines over the time/frame the action occurs at. This timeline also hosts a variety of animation controls, such as the record button, which automatically generates key frames at the current frame when the animator manipulates an object. The active time (the section

of the timeline played back when play is activated) can also be specified here, as well as the frame rate (the number of frames per second) and the current frame.

The bottom section of the screen is devoted to the Action Editor. In this, the user creates actions specific to an object. This is where separate character actions are created to be used by the animated model in the 3D game, like walk, turn, strafe, etc. This editor does not allow you to combine the actions of several objects into one action. Thus, when building a rig, this disallows the use of empty objects as targets for Inverse Kinematic solvers. Instead, as a bone is part of an armature object, use a disconnected bone.

The editor allows for large scale and small scale manipulation of key frames. Each animation event is indicated by a yellow or white diamond in the channel corresponding to the object within the action being edited. White indicates the action is not selected; yellow indicates it is selected. If a key frame is essentially a duplicate of another directly before or after it (same action is specified in both key frames) then they are connected by a purple bar as seen in fig. 8.

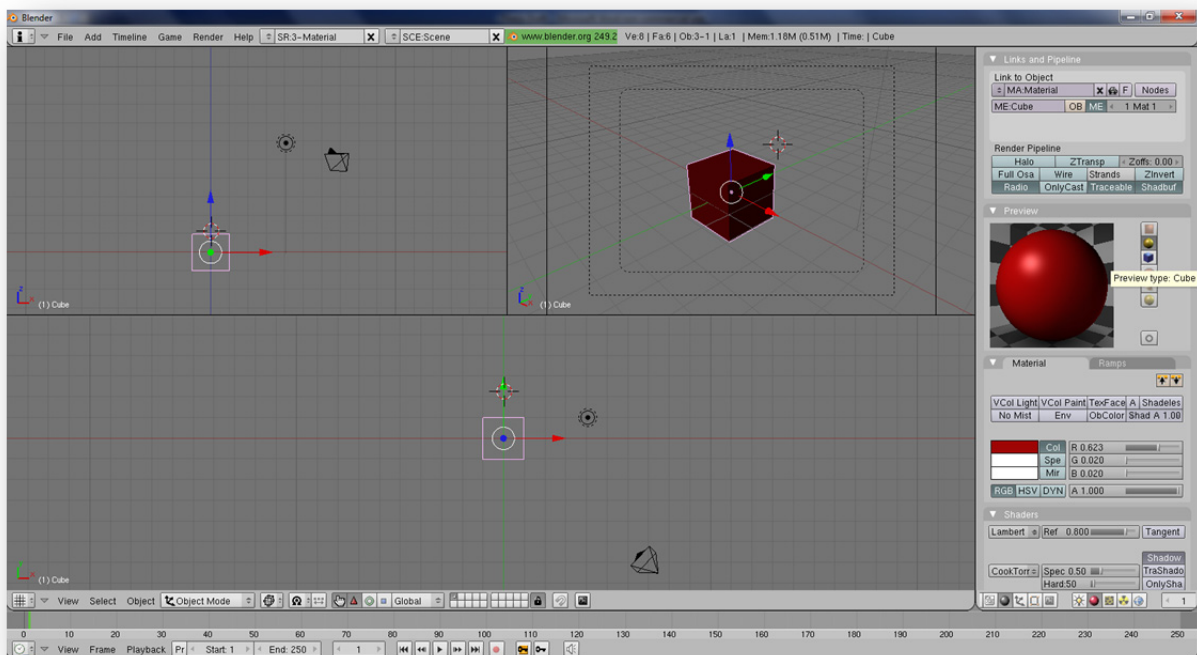


FIGURE 9: THE MATERIAL LAYOUT.

The material layout consists of three 3D viewport screens: front (orthographic), camera (if available, otherwise just perspective), and a wide top view (orthographic) below these two. Below these viewports is the timeline screen, allowing materials to be animated, and to the right is the buttons screen automatically set up to display the material tab.

Blender boasts its own game engine. It uses the Bullet physics engine to handle rigid body dynamics. For more straightforward game logic, Blender offers the Logic view of the buttons screen, which has three main data blocks: actuators, sensors, and controllers. Sensor blocks are capable of specifying that an event happen forever or at the click of a mouse. Controllers are typical logic operators such as AND and OR. Actuators specify an action

to take, from playing an animated action to switching the game to the next level. Sensor blocks are linked to Controller blocks, which in turn are linked to the desired Actuator blocks. These blocks are specific only to a single game object, so passing data around requires some creativity on the developer's part.

For the object Suzanne in fig. 10, we have added an "always" sensor, meaning the connected actuator will constantly be running. Linked to the sensor is an "AND" logic operation, which is linked to a motion type actuator, which is set to move 0.2 in the x-direction. So, this set of logic blocks will move Suzanne in the positive x-direction at a rate of 0.2 units per frame forever.

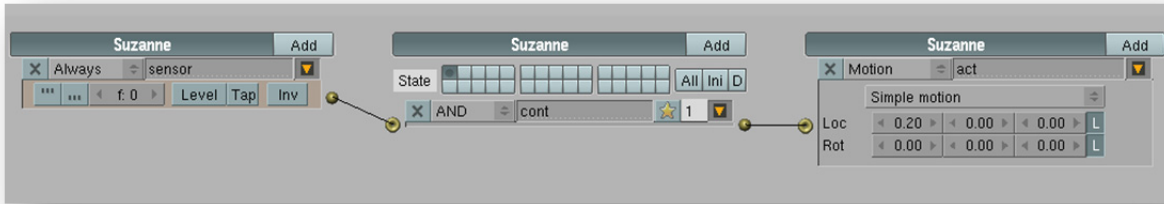


FIGURE 10: AN EXAMPLE OF A LOGIC BLOCK SETUP

Blender supports Python scripts, both in its game engine and in the program itself, but as this was not the programming language the project was focusing on, Blender could not be the primary development program.

2.2.1 MODELLING WALKTHROUGH

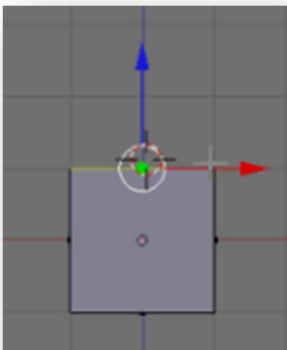


FIGURE 11: INITIAL CUBE WITH TOP FACE SELECTED

Most modelling begins with a primitive shape like a cube, sphere, or cylinder. After turning on Edit Mesh mode, a user can begin to manipulate the mesh on three levels: vertex, edge, or face.

Vertices can be manipulated through translating their position only; rotating and scaling has no effect on them. This is the level modellers work on in order to make fine changes.

Edges can be manipulated through translating, scaling, and rotating. This is the next level of detail up from vertex, as manipulating an edge manipulates the two vertices that define the endpoints of the edge, as well as the edges that connect those vertices to other vertices.

Faces are the highest level that can be manipulated in Blender. These are the visible sections of the mesh. They can be scaled, rotated, and translated.

All three components can be extruded from, though generally extruding on the face level is the most useful. Extruding a vertex produces no



FIGURE 12: TOP FACE EXTRUDED AND ROTATED

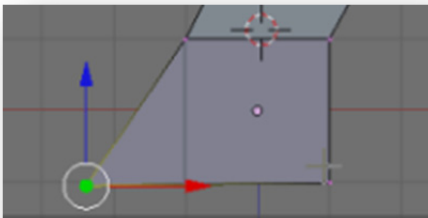


FIGURE 13: MANIPULATING A VERTEX

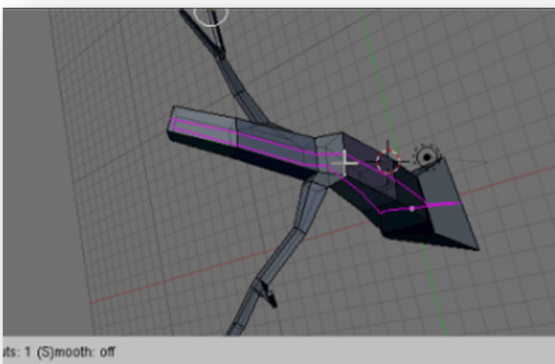


FIGURE 14: APPLYING LOOP SUBDIVIDE TO MESH

mesh, only another vertex. Extruding an edge creates a single face between the edge being extruded and the new copy created by the extrusion operation. Extruding a face creates a new face as well as faces to connect the new face to the edges that defined the old face. In this walkthrough, face extrusion is used extensively as the modelling strategy in use is extrusion modelling. The top face of the cube is selected and extruded. It is also rotated, as a tree's trunk is being built here, and the trunk is none too straight.

Extrusion is used to block out the crude shape of whatever the mesh will be. As in 3D games it is advantageous to keep the polygon count low in order to make the game easier to draw in real-time, extrusion modelling tends to produce acceptable models with low levels of detail if extrusion is used sparingly.

Instead, it is better to manipulate the mesh produced by minimal extrusion in order to best represent the target shape. Rotating, scaling, and translating, as seen to the left, are the operations that allow the modeller to manipulate as necessary.

Adding detail often requires that the blocked out shape be further refined with subdivisions. There are many ways to do this, but the ones that offer the most control are Loop Subdivide and Knife Subdivide.

Knife Subdivide allows the user to freely draw cuts across preselected edges of the mesh.

Loop Subdivide is a tool created to intelligently suggest possible cuts for a mesh depending on the location of the mouse over the mesh. It will follow the major line of the surrounding edges, either vertically or horizontally, generally, though organic meshes suggest far more complex lines. The edges suggested will stick with the established contours.

In the case of the tree, three possible

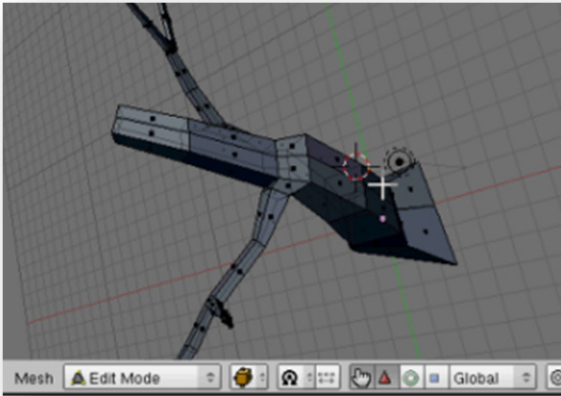


FIGURE 15: MESH AFTER LOOP SUBDIVIDE APPLIED.

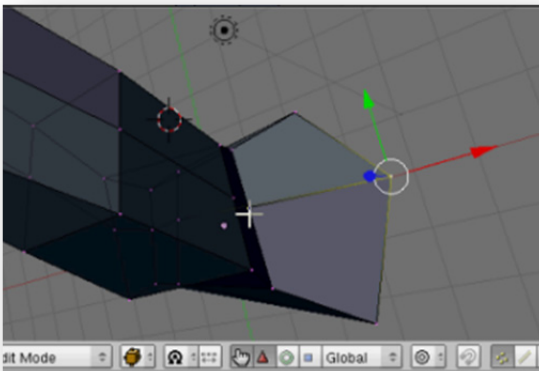


FIGURE 16: PULLING OUT VERTEX ON THE EDGE CREATED BY LOOP SUBDIVIDE.



FIGURE 17: MESH WITH SUBSURF APPLIED

options were suggested when the mouse was placed over different points of the trunk. Two vertical options were suggested, one as seen in purple in fig. 14; the other was down the trunk perpendicularly to the cut suggested in fig. 14. The horizontal cut suggested was between the two established “rings” of the tree trunk.

Fig. 15 shows the result of using the vertical cut displayed in fig. 14.

The newly created edges and vertices from the subdivision operation can now be manipulated to better detail the mesh. In this case, the cut creates a hexagonal shape for the tree trunk, rather than a square, which did not allow enough leeway for the circular nature of the trunk that needed to be established.

The last step is to add a subsurf modifier, which adds resolution to the mesh while still allowing the modeller to manipulate it on the low level it was built on. The resolution can be dialled up as necessary, added many more faces to give the mesh curvature, but when working on a mesh for a 3D game, at most the subsurf level should be dialled up to 2.

The view in fig. 17 displays the tree mesh created, as well as a glimpse of the effect the subsurf modifier has upon the finished product, allowing the modeller to work on the low level while seeing the changes made on a higher level of detail that the finished model will be added to the game at.

2.2.2 ANIMATING WALKTHROUGH

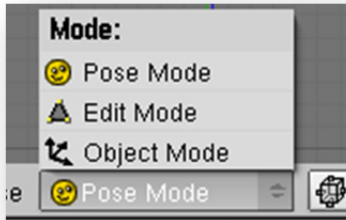


FIGURE 18: SELECTING POSE MODE IN A 3D VIEWPORT CONTROL PANEL

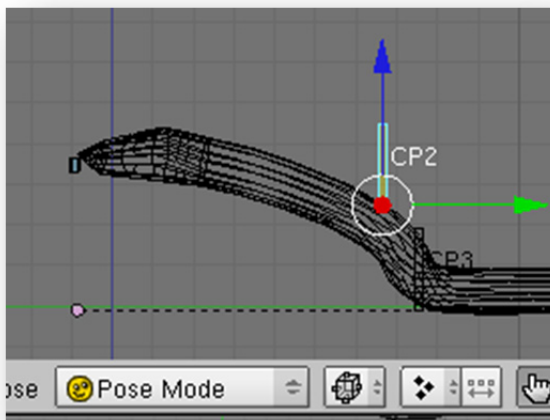


FIGURE 19: TRANSLATING A BONE POSITION

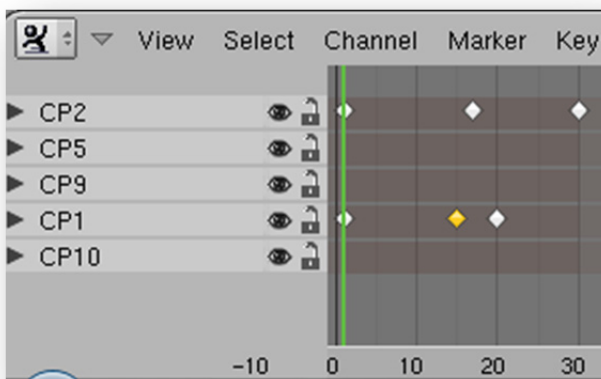


FIGURE 20: ACTION EDITOR SCREEN

During the initial stages of animating, the goal is to block out the major motions by moving along the timeline and moving the major bones into position, generally foot and hand target bones. Because most animations will be looped within the game and the mesh is not actually moving on its own, the motions must be blocked out in a “slide along” fashion while the mesh stays rooted in space.

The armature object must be in pose mode for the bones to be animated.

Bones can be manipulated through translation and rotation, primarily. Scaling can be used, but it isn't recommended unless it is necessary, as it may cause odd changes in the mesh and bones during other sections of the animation if not handled carefully.

In fig. 19, a bone is being translated from its originally recorded position. This creates a key frame, which will in turn create IPO curve data and action data for the armature object for that specific bone. The IPO data created will be in channels specific to what was manipulated, in this location change in the z-axis.

Curves can be edited dragging the key frame or Bezier handles that control how the interpolation between two key frames is handled.

The level of detail offered by the IPO Curve Editor screen can be daunting and syncing the animations of different objects is impossible, so the Action Editor screen comes in, as seen in fig. 20. Key frames can be moved, duplicated (individually or in groups), and scaled (in groups, around the green line marking the time currently being edited).

2.2.3 APPLYING MATERIALS WALKTHROUGH

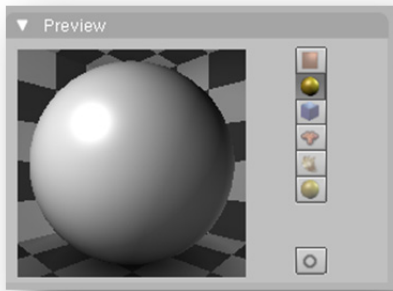


FIGURE 21: THE PREVIEW WINDOW

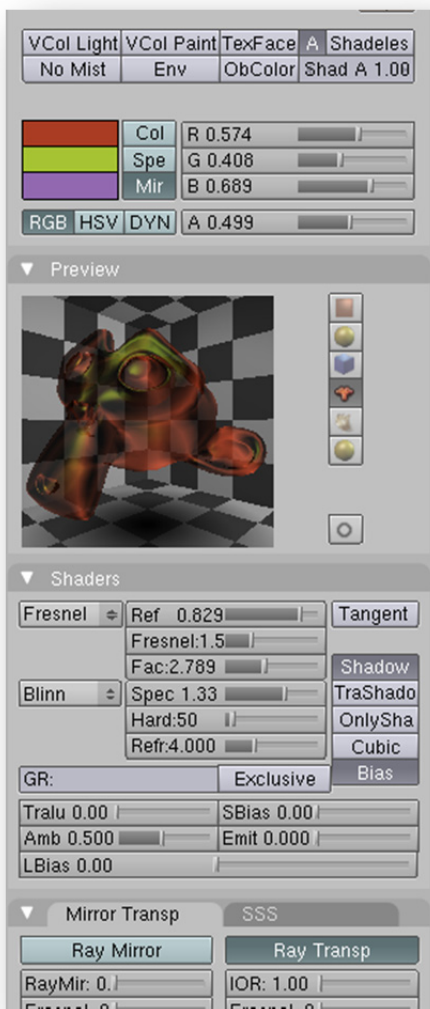


FIGURE 22: THE MATERIAL, PREVIEW, SHADER, AND MIRROR TRANSPARENCY BUTTON GROUPS,

Of all the tools available in the buttons screen when working with materials, the preview window is by far the most intuitive. It can be customized to display the material on a sphere, as seen in fig. 21, a cube, on Suzanne (the irregular shape all 3D studio software solutions provide just for testing out textures and modifiers – a monkey head in this case), on a particle system that looks rather like grass, or in the material's natively flat plane state.

The most basic materials can be created in the Material buttons group, seen in at the top of fig. 22. Here, Shading can be turned off, as well as mist effects and alpha effects.

Col is the main colour the material will be, which can be specified in one of two colour models: RGB and Hue Saturation Value. Spec is the specular highlights that a material will gain when hit by light. Mir, which is Mirror, which describes how the material will distort images reflected off its surface colourwise.

Alpha can affect the transparency of the material, as seen in fig. 22. Note how the A button between the TexFace and Shadeles buttons is activated, how A beneath the RGB sliders is dialled down to 0.499, and how Ray Transparency is activated in the Mirror Transparency button group. Note how this affects the preview: the checkered background is showing through Suzanne.

Visible in fig. 22 is the Shaders button group. Currently, Fresnel and Blinn are being used—Fresnel for shading and Blinn for specular. There are many different shaders available that produce varied effects. Phong, for instance, is particularly good for creating metallic highlights.

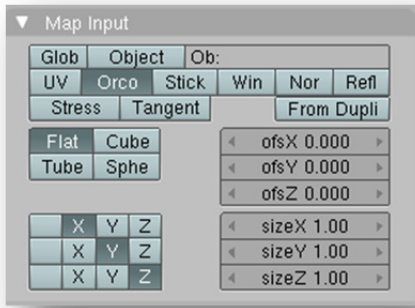


FIGURE 23: MAP INPUT BUTTONS GROUP

Often, more than just the options described above are required to get just the right look. Textures are thus needed to add layers of complexity to a material. Most 3D games rely heavily on drawn textures with limited effects added to them in order to reduce the burden of drawing the textures in real time. Thus, most textures are image textures.

Mat Input (fig. 23) specifies how the texture is mapped to the mesh. Flat, Cube, Tube, and Sphere are some options that will specify how a texture is mapped onto a mesh that may not have the right UV coordinates for the texture. For instance, if a sphere was lacking the proper UV coordinates, selecting sphere would ensure that an unwrapped/stretched out picture of the surface of the earth was properly applied to the sphere.

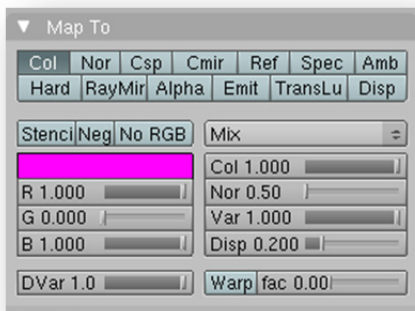


FIGURE 24: MAP TO BUTTONS GROUP

Map To contains buttons that will specify how textures effect certain channels. For instance, a black and white image could be used specifically on the Alpha channel to specify a particular way the object needs to become transparent.

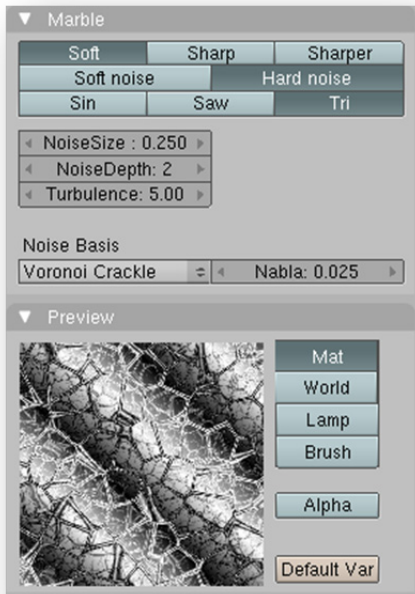


FIGURE 25: AN ALTERED MARBLE TEXTURE.

In the Texture buttons group, the texture can be named and its type specified. Type options include those depicted in fig. 26.



FIGURE 26: TEXTURE TYPE OPTIONS

By choosing an option like Marble, two button groups become available: Marble and Preview. Marble allows the look of the texture to be altered as necessary.



By mapping the texture in fig. 25 to the Alpha channel of the material, the look of the transparency is changed drastically.

FIGURE 27: TEXTURE APPLIED TO ALPHA (TRANSPARENCY) CHANNEL.

2.3 XNA IN VISUAL C# 2008 EXPRESS

XNA is not a game engine in itself, but rather a free collection of classes that allow for combining any elements desired (music, sound effects, meshes, cameras) in a format that can be played on a Windows operating system or on an Xbox gaming console. It has no physics engine, but allows for a coder to have the fine control desired over every element. It can be augmented with physics engines available online for free.

The project used XNA version 3.1, which is compatible with Microsoft Visual Studio 2008 and Microsoft Visual C# 2008 Express. The latter is available for free download on Microsoft's website and through MSDN, while the former is very expensive. XNA boasts good documentation, both offline and online, as well as a host of pre-packaged tutorials with precise instructions and explanations.

Its major drawback is its lack of viewport: all elements have to be placed and worked with according to vectors, which can be very unintuitive for those that are very visually oriented. It was hard to correctly position a camera without the visual feedback.

2.4 UNITY

Unity recently released a free version of its software. It boasts as many viewports as the user wants (so it is possible to set up the traditional four view (top, right, front, and perspective) and is compatible with whatever system the user is willing to figure out how to compile for, from PC to iPod. It imports Blender meshes very well through the FBX exporter (compliments to Autodesk) and works with Visual C# Express Edition should the user not want to write C# scripts in its native script editor. It is unfortunately a big program in terms of RAM and processor usage, but with the right system this is not a big drawback.

Where Blender did not offer enough control over the game and XNA left far too much up to the programmer without enough visual support, Unity in conjunction with Blender for visual elements and Visual C# Express for code editing seems to offer the right amount of control and visual context at the right price tag.

Visual Studio speeds up working with Unity as Unity's debugger can jump right to the errors it finds in scripts via the hooks that Microsoft did away with in Visual C# Express. Also, Unity and Visual Studio can create a project file together, making it possible for Visual Studio to error check code before debugging. Debugging manually must be done in the MonoDevelop IDE that is provided with Unity though, in order to step through code properly.

Unity comes with the advantage of tons of resources, from scripts from example projects to many tutorials. It is possible to adapt many native scripts, translating from JavaScript to C# when necessary. Scripts in Unity can have their own user interface, making interacting with scripts a potentially visual process.

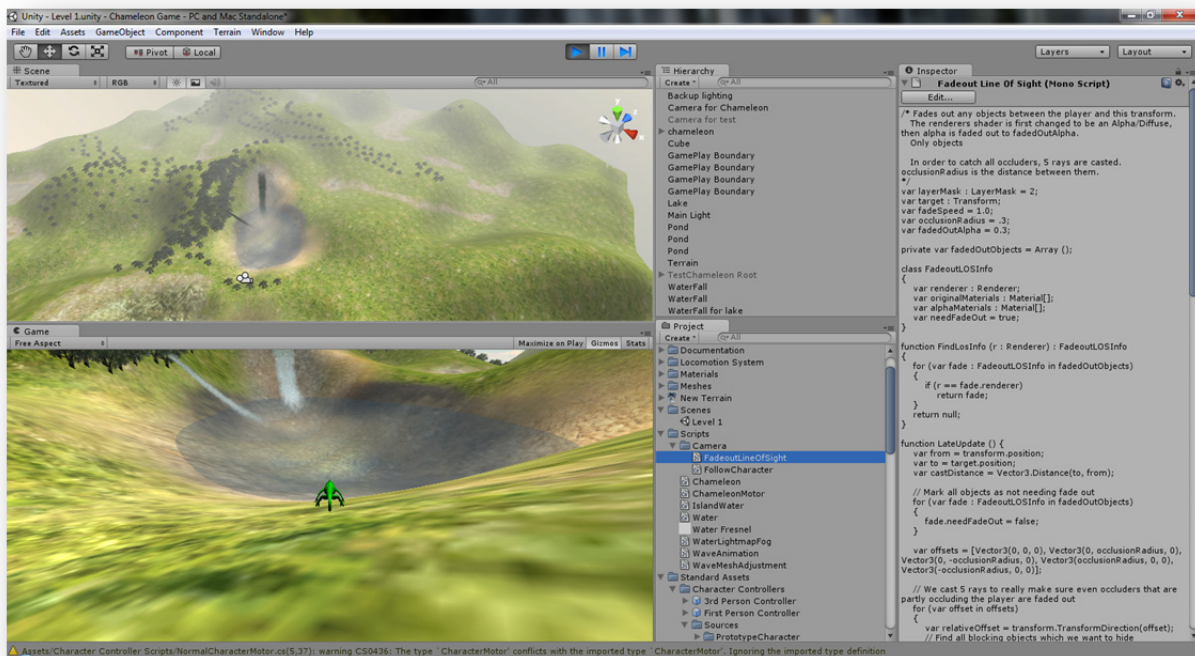


FIGURE 28: UNITY WITH VIEWPORT (TOP LEFT), GAMEPLAY SCREEN (BOTTOM LEFT), HIERARCHY VIEWER (MIDDLE TOP), PROJECT VIEWER (MIDDLE BOTTOM), AND INSPECTOR WINDOW.

2.5 VISUAL STUDIO 2010

Visual Studio 2010 is an integrated development environment for programming in a variety of languages. It can format and check the syntax of a variety of languages, including C# to JavaScript. It can perform error checking if a project file is created referencing the packages containing the classes being used. With the project file, it can also make suggestions based on the contents of the class being referenced.

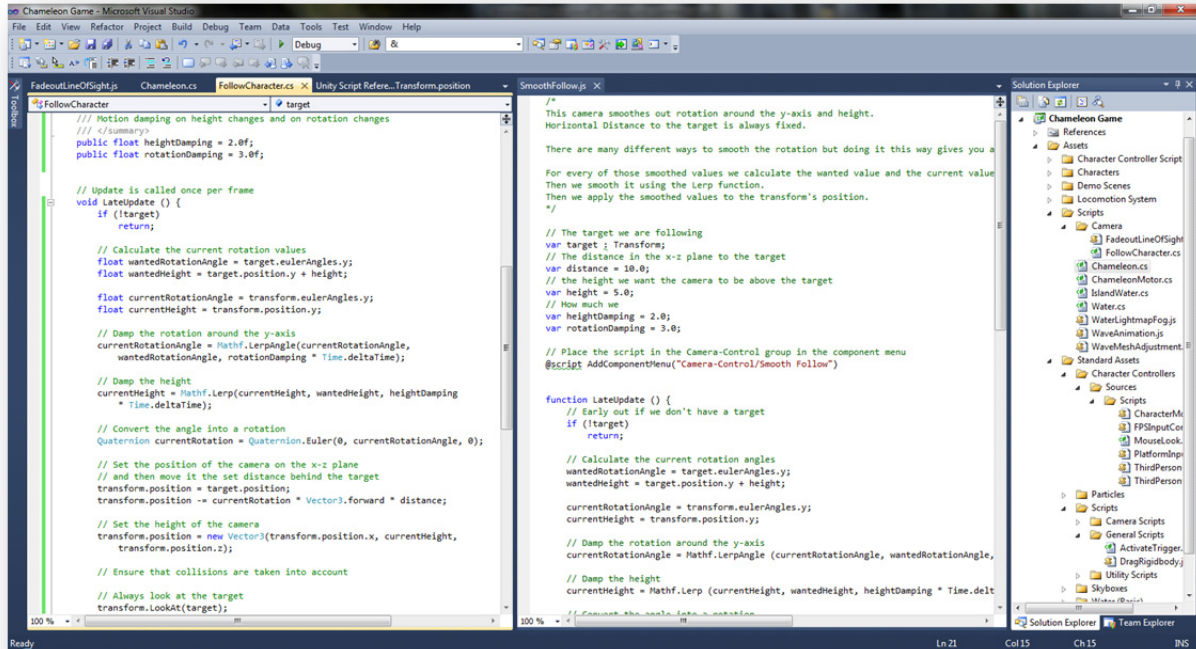


FIGURE 29: VISUAL STUDIO 2010 WITH SIDE-BY-SIDE EDITORS RUNNING. TO THE LEFT IS THE SOLUTION EXPLORER, WHICH DISPLAYS ALL PROJECT FILES AND FOLDERS. CURRENTLY, A NATIVE JAVASCRIPT FILE IS BEING TRANSLATED INTO C#.

2.6 GIMP

GIMP is an image editing program comparable to Adobe Photoshop. It supports layers, alpha channels, and smart selection based on colour. It has the standard tools: sharpen, blur, paintbrush, ink, eyedropper, smooth, smudge, paint bucket, and various selection and manipulation options. It can support tablet input, altering brushes based on tablet pen pressure and pen tilt. It can output images in a variety of formats, including .bmp, .png, .jpeg, and .gif.

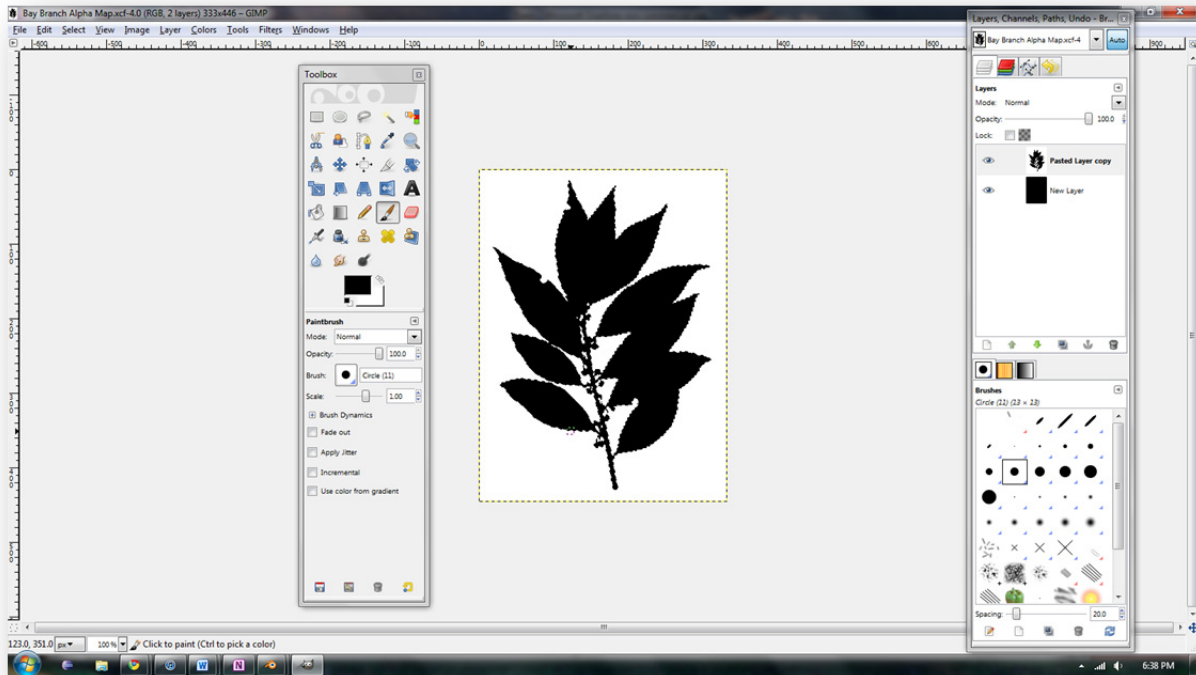


FIGURE 30: CREATING AN ALPHA MAP IN GIMP

3 CHAMELEON GAME

3.1 CONCEPTION

Conception requires little in the way of software, at least initially. First, there must be an idea. This idea must be tailored to fit the confines of a game in that it must have a goal, it must have some aspect that makes it worthwhile to replay, and there must be some obstacles present to make gameplay interesting. For this project, the goal was to have a chameleon survive his journey to obtain the antidote to the strange poisoning that causes him to change colour rapidly and gradually lose life. The ideas explored to make this game worth playing again were having the terrain be randomly generated, having the goal and the starting position randomly shift per session, and have the enemies' positions not be fixed in any way, making the route's hazards unpredictable. Snakes were added as obstacles, which would detect the chameleon's presence if he did not blend with his surroundings properly or he moved too quickly through their field of awareness, and the effect of the poison on the chameleon's health essentially created a time limit.

3.2 DESIGN

Design is a more expensive process than conception. Concept art is often done in Photoshop or a similar program, which are all on the range of thousands of dollars for a professional copy and hundreds for a student copy, which doesn't permit any sort of commercial work. The closest freeware available is GIMP, which is more of a very basic copy of Photoshop, allowing work with layers and textures. Design not only covers the artistic aspects of the game; code design also begins here. Picking user controls (left, right, up, down) and outlining the specifics of game play begin to set out the most basic coding aims.

3.2.1 CHAMELEON

The initial outline for the chameleon character was simple: he needed to be fully mobile in a 3D environment and he had to be able to change colour seemingly randomly.

Over the course of research of various chameleons, his real world reference became a young veiled chameleon, as they had the simplest shape. Searching through the images on www.deviantArt.com was better as they usually have the highest quality photographs to act as a reference image.



FIGURE 31: SIDE VIEW OF YOUNG VEILED CHAMELEON USED AS REFERENCE IMAGE DURING MODELLING

The standard WASD key configuration was decided upon to be used (W is forward, A is left, S is backward, D is right), as well as the arrow keys. Left and right would produce a sidestepping, strafing, motion. At this point, the mouse provided rotation. Later experimentation caused the revision of this strategy to instead have the Q (or 4 on the keypad) and E (or 6 on the keypad) keys be left and right rotation respectively while the mouse was used to control the angle of the camera, allowing it to orbit the chameleon so the player would have more control over his view of his surroundings since sight is what winning the game depends on. With the inclusion of a duck action, another key had to be added, so the shift keys were designated to handle triggering that action.

As the chameleon is supposed to be able to climb, there needed to be some method of detecting his collision with a tree trunk and then allowing him to interact with the tree's surface as though it had suddenly become the ground. The same system would need to be applied to every branch on the tree.

The chameleon's objective is to come into contact with the mesh that represents the antidote to the poison he ingested prior to the game's start. This triggers the end of game sequence.

The effects of the poison acts as a timer; the player has only so much time to track down the antidote or have the chameleon die in front of him. The random colour changes at random intervals also acts as a force the player must work against as if the chameleon is standing in front of a brown tree trunk and turns purple, he will be more easily detected and potentially eaten by snakes.

3.2.2 SNAKE

The snakes would be placed randomly every game to increase the re-playability factor so every game instance is unpredictable. Every snake has a "dome of awareness" that once the chameleon lies within makes it vulnerable to attack. Actions such as moving too quickly and moving in front of objects when the chameleon is the wrong colour alert the snake, making it attack. If the chameleon ducks or moves when attacked, it has a chance of avoiding the attack, but otherwise it is dead in one strike.

3.2.3 ENVIRONMENT

The environment had to be quite large or the task of finding the antidote would be too easy. Water and foliage would be included to make interaction with the environment more interesting.

3.3 MODELLING

3.3.1 CHAMELEON MODELLING

The chameleon began with a cube and the reference image seen in fig. 32. The initial hours were spent extruding faces in order to correctly block out the chameleon's profile, as arms and legs are complicated and would come later.

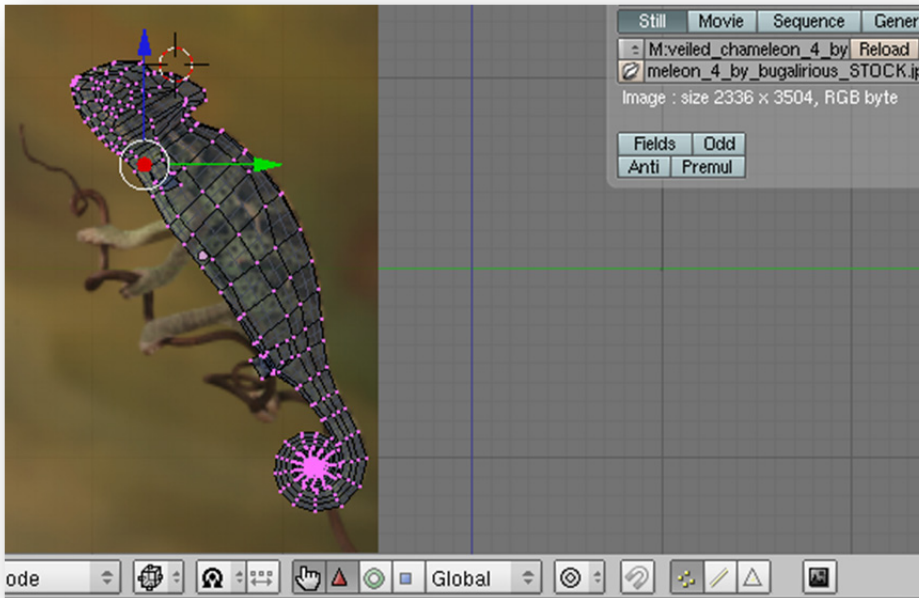


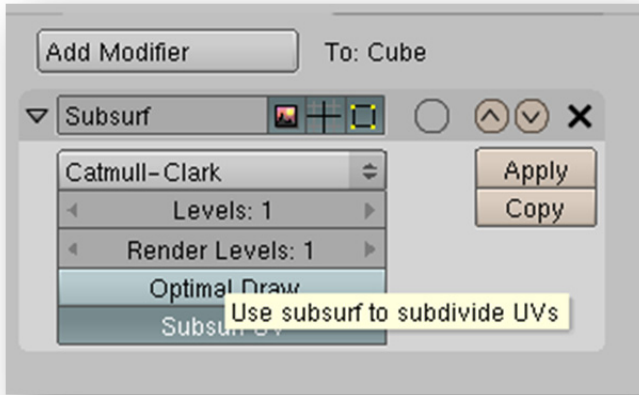
FIGURE 32: BACKGROUND REFERENCE ADDED IN SIDE VIEW. BUILT MESH ACCORDING TO IT.



FIGURE 33: LEG IN PROGRESS. ARM MOSTLY FINISHED

Once the profile was suitably developed, adding detail to the interior was done by adding cuts along the side of the body with Loop Subdivide. As seen in fig. 33, three loop cuts have been added. The vertices were adjusted to better capture the shape of the chameleon, and then the faces where the arm and the leg were to come out were extruded as necessary to shape the limbs. More detail is necessary in joints in order for the mesh to deform appropriately when rigged and animated, so more cuts had to

be made around the hip, knee, elbow, and wrist joints in order to ensure that there was enough detail at these points.



A subsurf modifier was applied, as the subdivision modelling strategy was being followed. The low resolution mesh was still being manipulated, but the mesh was made to render and display as though it was one level of detail above that.

FIGURE 34: SUBSURF MODIFIER APPLIED TO SMOOTH THE LOW RESOLUTION MESH.

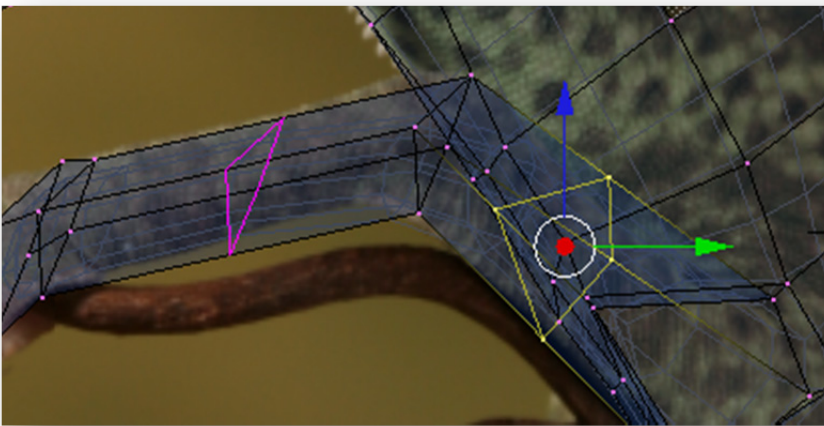


FIGURE 35: ADDING CUTS TO THE LEG (PINK - IN PROGRESS, YELLOW - JUST ADDED).

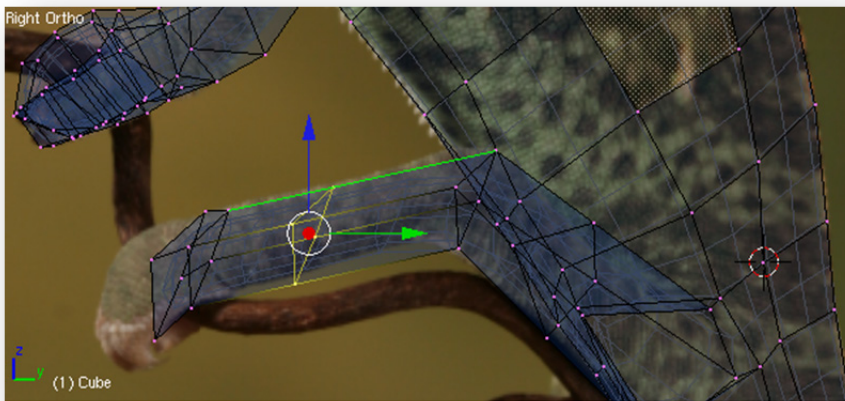


FIGURE 36: ADJUSTING WHERE THE CUT WILL OCCUR ON THE LEG ALONG THE GREEN EDGE.

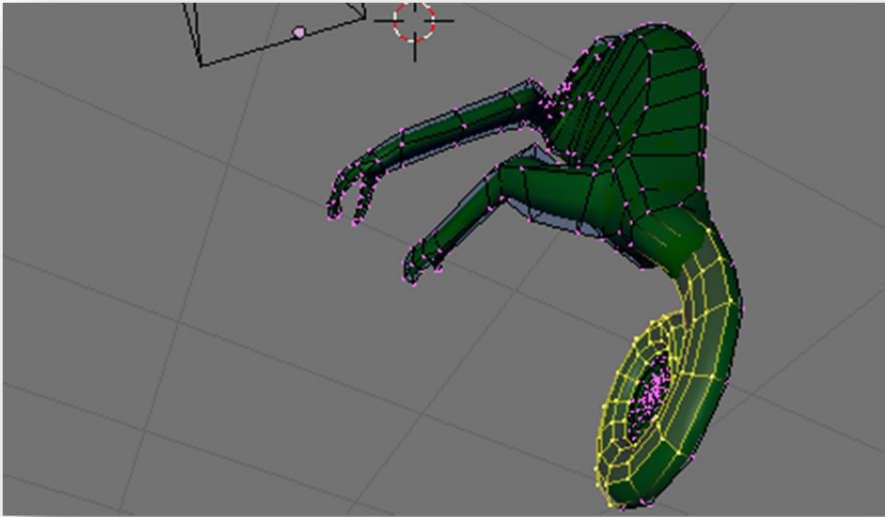


FIGURE 37: THINNING THE TAIL RELATIVE TO THE WIDTH OF THE BODY.

Once the detailing of the hands and feet was done, the tail and the torso had to be adjusted again to ensure that the proportions were correct for when the half of the mesh was mirrored and fused together to form the two halves. Building one half of objects with bilateral symmetry is a common tactic to reduce the amount of modelling necessary.

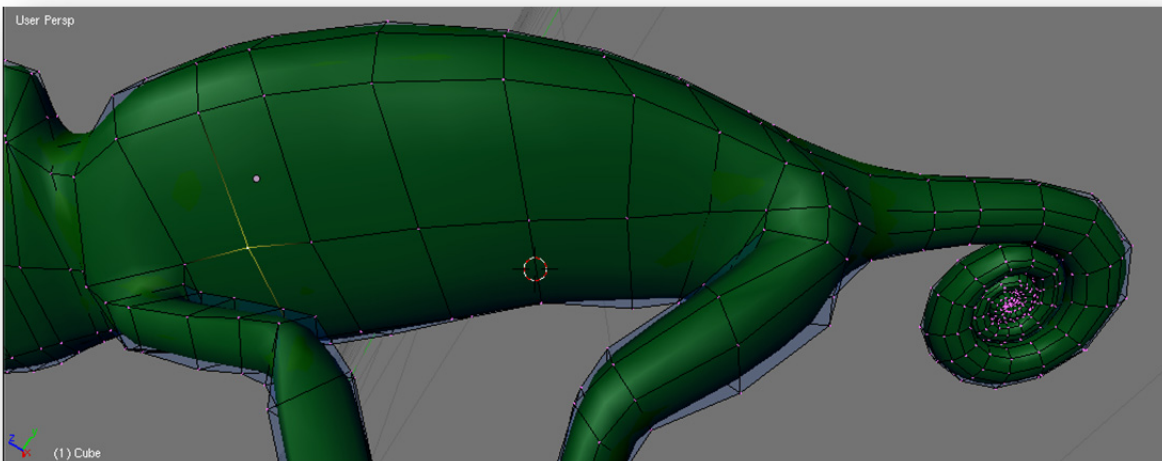


FIGURE 38: ADJUSTING THE WIDTH OF THE TORSO.

Fine detailing could now be done, particularly around the face. The eye, formerly just a square face, was adjusted with cuts and by moving the vertices to create something a little rounder before efforts were made to make the eye bulbous. The Knife Subdivide tool was necessary here, as seen in fig. 39, in order to correctly give the eye shape.

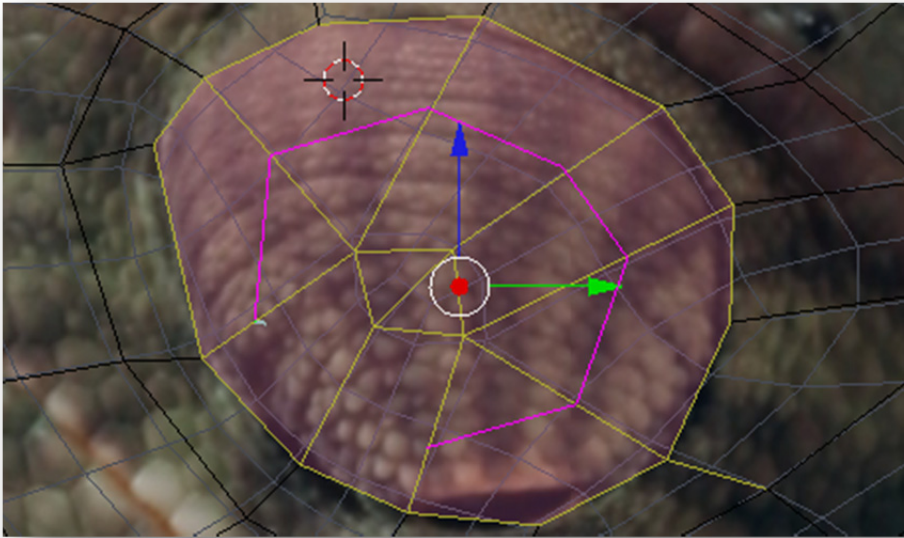


FIGURE 39: CUTTING THE EYE SO IT WILL BE ABLE TO HAVE THE CORRECT "DOME" SHAPE. PURPLE – CUTS IN PROGRESS.

Models must always be built in the “rest position” – the position that has every joint as relaxed as possible so the mesh is not influenced oddly when it is animated. Because the initial model was not built this way, problems came up with the tail curled up (it looked awful when Unity imported it and played the tail uncurling animation) and the how the hands were built at the wrong angle and closed, making them look odd when spread during the walk cycle.

So, the tail was remodelled, as seen below. Re-rigging it posed some challenges as getting the bones to not overlap incorrectly turned out to be quite a process. Thus, building it right the first time is highly advisable; do *not* redo any rigging unless it is absolutely necessary.

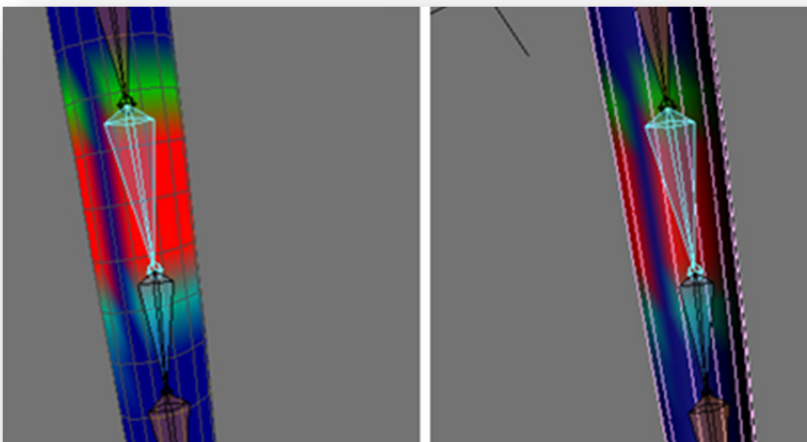


FIGURE 40: NOTE THE GAPS IN THE COLOURS (PAINTED WEIGHTS). USING THE PAINTING MASK TOOL (RIGHT) HELPS HIGHLIGHT MISSED VERTICES VERSUS USING THE NORMAL WEIGHT PAINT VIEW (LEFT).

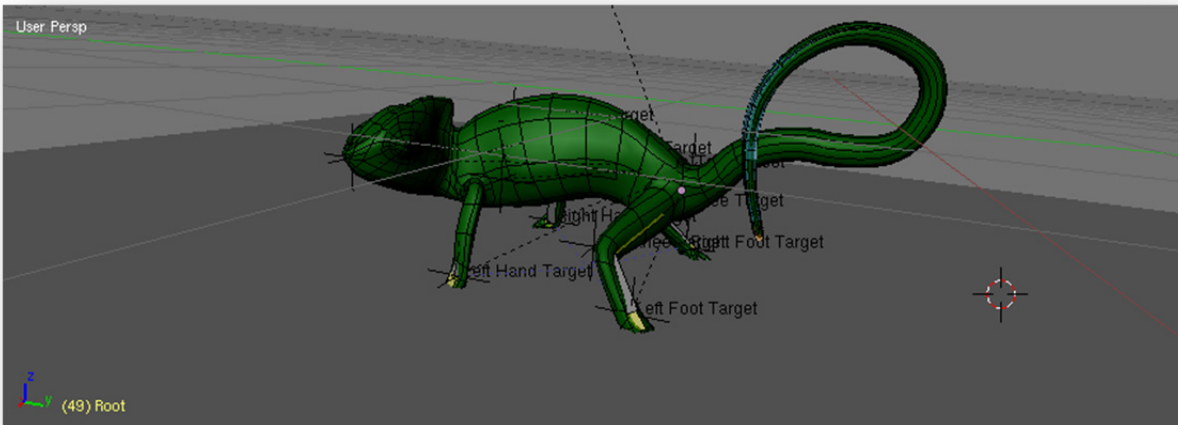


FIGURE 41: FIXED TAIL. IT CURVES NICELY NOW.

3.3.2 SNAKE MODELLING

Starting with a cylinder, one of the premade meshes provided by Blender, the snake model was quite simple to generate. After laterally subdividing the cylinder and reshaping the rings, the snake shape was produced. A decimate modifier was applied to reduce the number of polygons and some clean-up work was done after that modifier was applied permanently, fusing vertices and cutting new edges to ensure the lower resolution mesh would bend properly. A subsurf modifier was applied to make the mesh appear smoother.

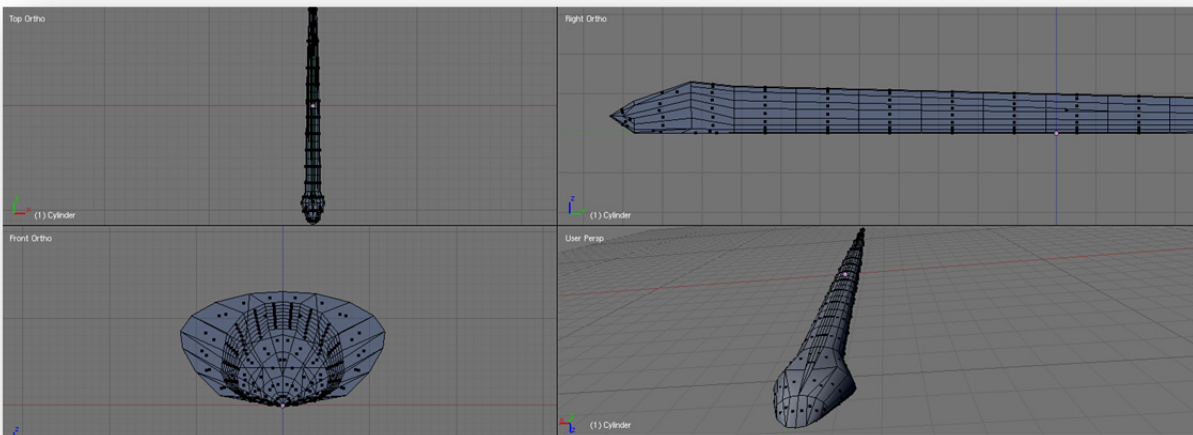


FIGURE 42: FINISHED SNAKE MODEL VIEWED IN POLYGON EDIT MODE.

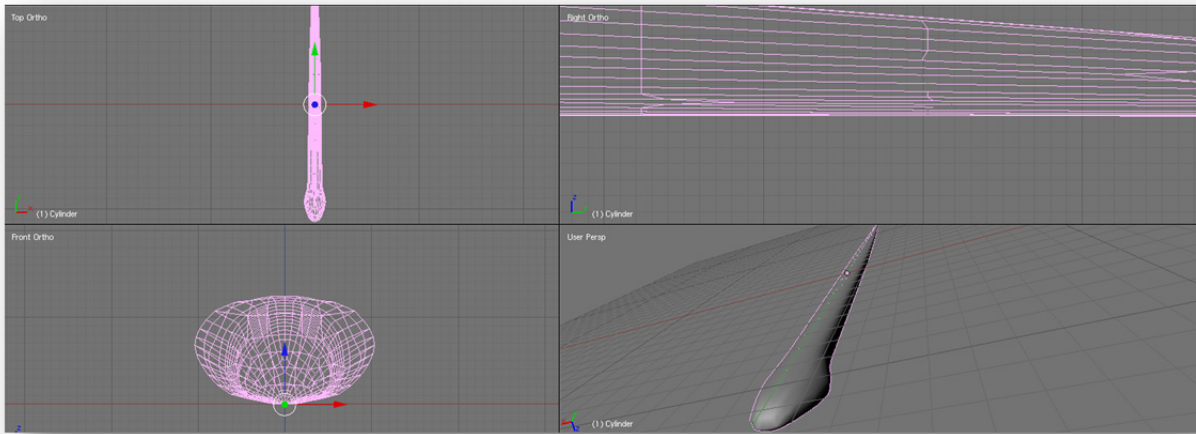


FIGURE 43: FINISHED SNAKE MODEL WITH SUBSURF MODIFIER APPLIED.

Building the rig was a more involved process. Research into creating a flexible spine that would require minimal manipulation during animation was done after using a spline as the spine proved impossible to import into Unity. A rig with 74 bones was developed, with 32 serving as the spine, 32 serving as Inverse Kinematic targets for their corresponding spine bone, and 10 serving as the control bones that would be manipulated during animation.

The spine bones were built, as seen in fig. 44, inside the snake mesh, flush with the ground. The IK target bones were built just above the snake.

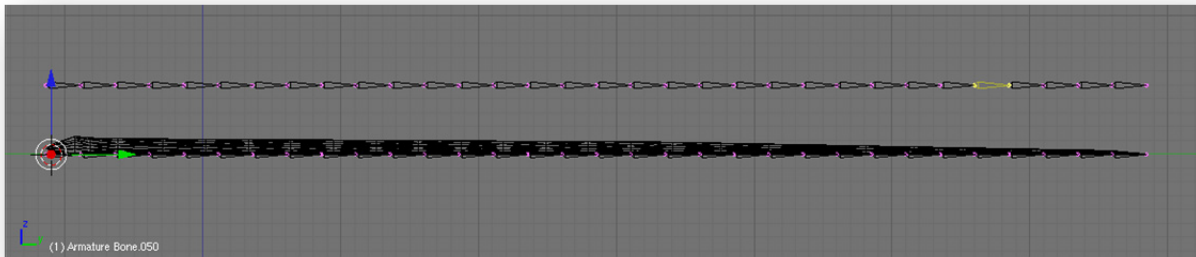


FIGURE 44: CREATING ARMATURE BONES. THE BONES THAT WILL BE DEFORMING THE SNAKE ARE THE ONES LYING AT THE BOTTOM OF THE SNAKE MESH.

The control bones were carefully added to bone spine and IK target chain. One control bone became the root bone of the spine, as is depicted in fig. 45. The other nine control bones were worked into the IK target chain at intervals; they were made parents of the bones following them in the chain and children of the bones directly preceding them, as is seen in fig. 46.

With this hierarchy established, the IK target chain and its control bones were moved farther up and the IK constraints were applied to each spine bone. This caused the spine chain to curve up towards the IK target

chain as each spine bone tried to point at its target IK bone, as seen in fig. 47. The control bones were also made to resemble spheres at this point to make them easier to select.

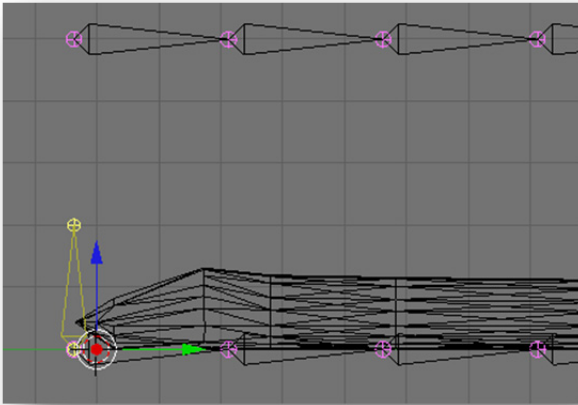


FIGURE 45: ADDING THE FIRST CONTROL BONE (YELLOW) TO THE SPINE BONE CHAIN.

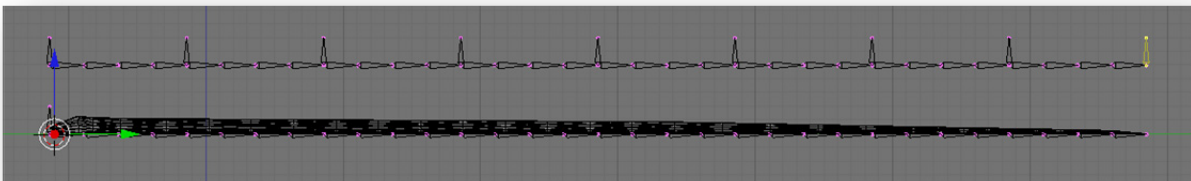


FIGURE 46: ALL CONTROL BONES IN PLACE ALONG TOP ROW.

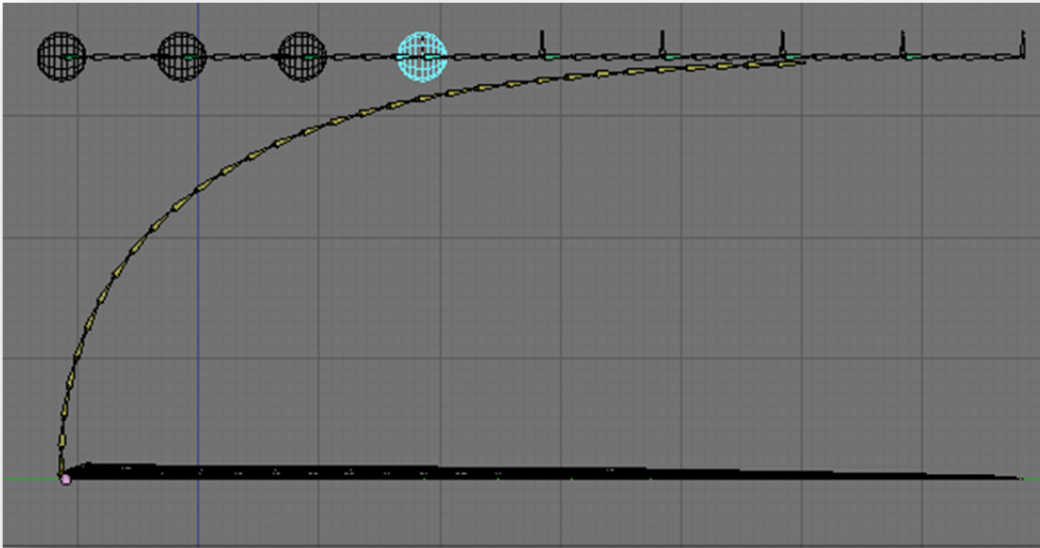


FIGURE 47: COMPLETED CONSTRAINTS: YELLOW IS IK SOLVER (SPINE BONE POINTING AT ITS CORRESPONDING IK BONE), GREEN IS STRETCH TO, WHICH POINTS AT THE NEAREST CP BONE IT POINTS AT (HALF ARE CIRCLES HERE).

The IK bones just after a control point have a constraint added to them called Stretch To. This permits the entire chain of four bones from just after a control point bone to the last bone before the next one in the hierarchy chain to stretch to touch the specified target. This is illustrated in fig. 49.

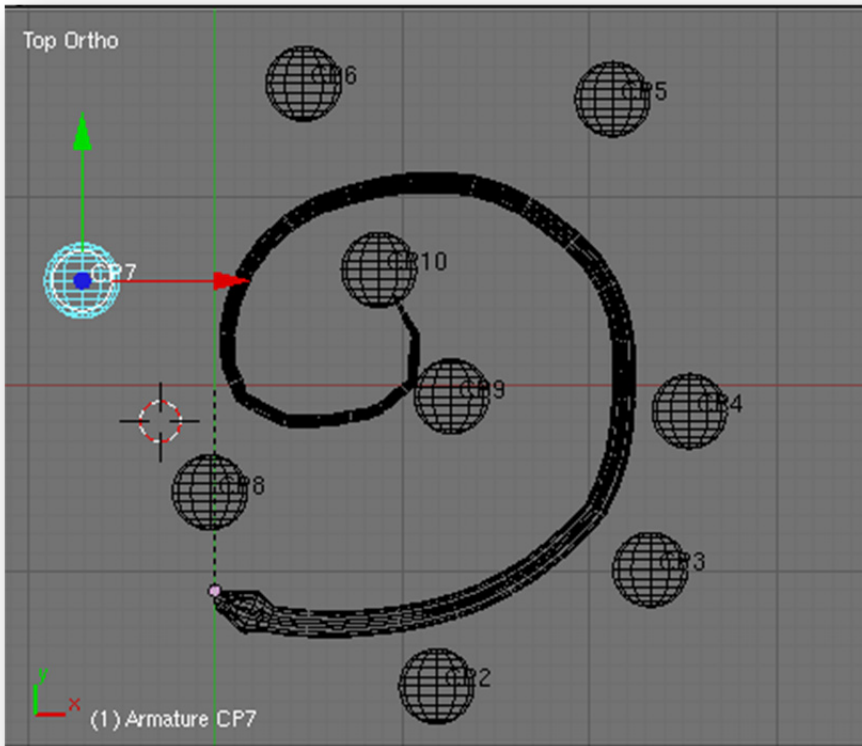


FIGURE 48: JUST CONTROL POINT BONES VISIBLE. MESH IS COMPLETELY RIGGED.

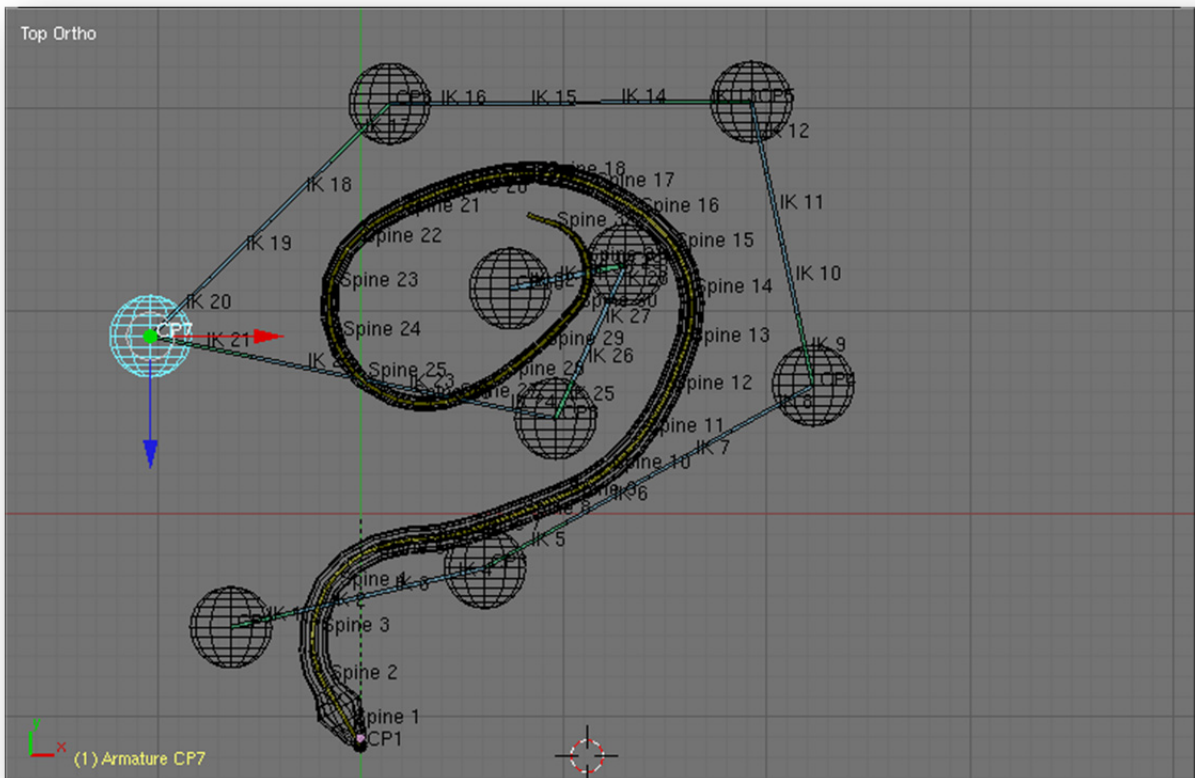


FIGURE 49: RIGGED WITH SPINE HAVING B-BONES WITH THREE SEGMENTS EACH

3.3.3 TREE MODELLING

Starting with the standard cube in Blender, by cutting, subdividing, and extruding, this tree was produced. A subsurf modifier was applied to give the tree trunk and branches the necessary curvature. Planes with complex transparency maps were applied to give the tree the twigs and leaves necessary for it to look alive.

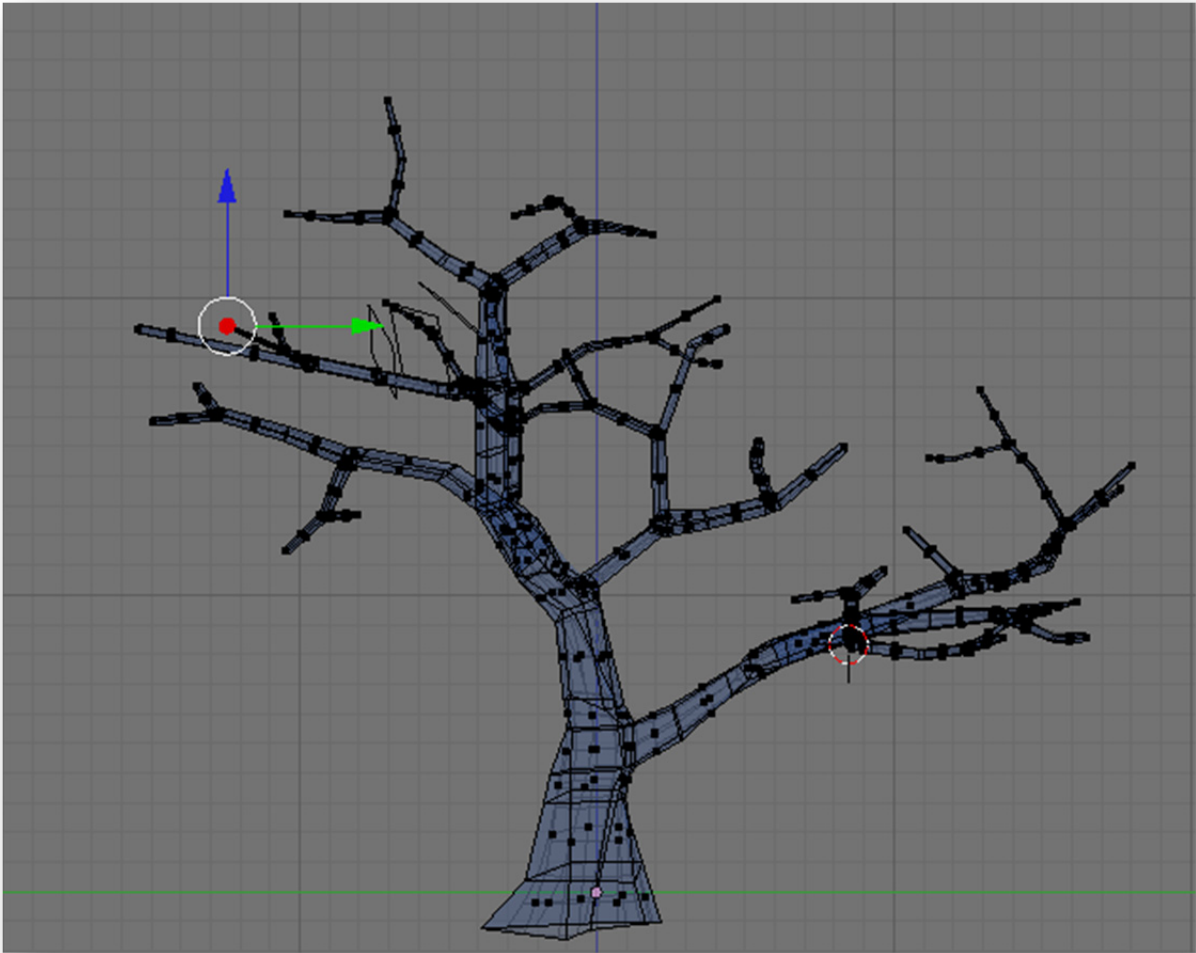


FIGURE 50: FINISHED TREE WITHOUT SUBSURF MODIFIER.

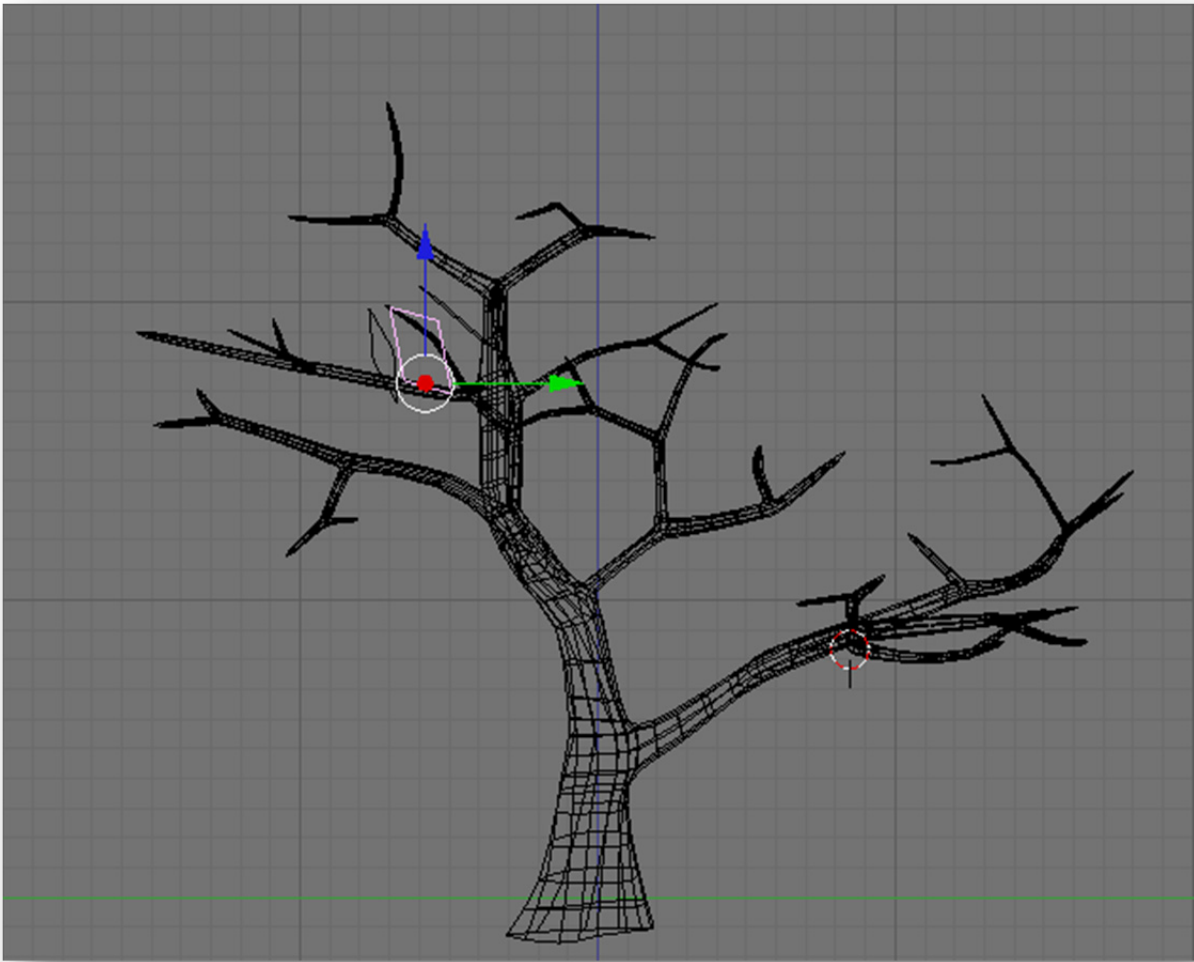


FIGURE 51: FINISHED TREE WITH SUBSURF MODIFIER APPLIED. NOTE THE PLANES ON THE BRANCH (PINK – SELECTED). THESE HAVE A LEAFY TWIG IMAGE APPLIED TO THEM.

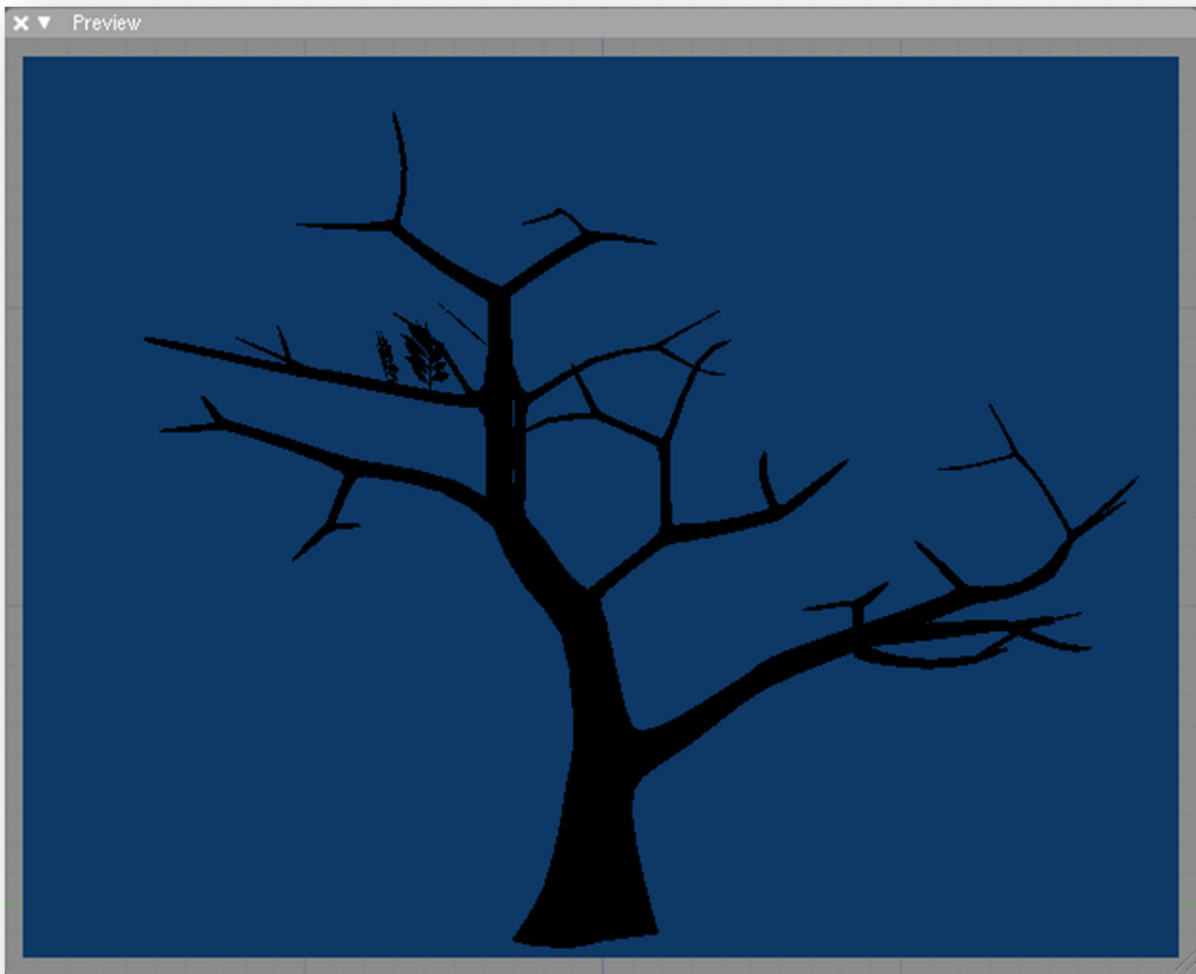


FIGURE 52: FINISHED TREE PREVIEWED WITHOUT LIGHTS.



FIGURE 53: TEXTURE TO SIMULATE LEAFY TWIGS

3.4 ANIMATING

Animating was done in Blender, though it can also be done in Unity. Trying it in Unity is not recommended as it is much easier to handle rigging and animating within Blender, which boasts more extensive controls and constraints than Unity.

3.4.1 CHAMELEON

For the chameleon, the greatest number of animation clips was required. Two walk cycles, one specifically for narrow branches, were developed. Turn animations were developed, but they are currently under revision as they are much too long to blend correctly back into the walk cycle when the player is executing quick course corrections. Two tail motions were developed: a tail curl on top and underneath, just so they could be played at random during walk and idle. A crouch animation was created to be played upon using the Shift key and when the chameleon is falling. A dying animation was also created to be played when the chameleon dies; it simply has the chameleon collapsing to one side.

3.4.2 SNAKE

The snake only required three animations: slither, idle, and strike. Of these three, slither was the most time consuming. It was based on a sinusoidal wave and was carefully tweaked to better mimic the actual motion of a snake as following the wave at a constant rate caused odd-looking slow down while the snake's head and upper body went through the peak and valley of the curve. These sections were compressed to better give the sense that the snake was gaining speed through these turns.

3.5 SCRIPTING

There are three options with animated meshes: continue in Blender or move into Unity or XNA through the free Visual Express C#. All have pros and cons. Blender's game engine is essentially platform independent: it will run on any operating system so long as the Blender engine is included with it. However, it is not a very good game creation route for anyone not familiar with the Python programming language. It allows for fairly complex GUI-based programming of elements and its physics engine is fairly capable of handling hair, cloth, and gravity, but there are issues, such as getting stuck on flat surfaces for no apparent reason.

With XNA, the lack of support for blending animation and handling collisions makes it a poor choice for continuing development in.

3.5.1 CHAMELEON

The major scripts the chameleon needed were ones that would connect his motion to the keyboard input, that would play the correct animations and blend them when necessary, that would change his skin colour at

random to random colours, and that would allow him to move correctly over the terrain, including obstacles such as rocks and trees.

Through the use of the character controller, a specifically adapted capsule collider with methods necessary to move a character, and Unity's input mapping, it was simple to write a script that would apply forward and lateral velocity as necessary, as well as applying the appropriate rotation. Handling the speed was a little trickier, but adapting some JavaScript found in a tutorial into C# worked well enough at applying gravity and handling collisions with the ground. For a time, it also applied acceleration according to the slope the chameleon was travelling on, but this was found to be unrealistic and more of a hindrance than an interesting game feature.

The negative aspect of using the character controller is that it imposes a limit on the slope it can travel up; the fixed maximum value is 180 degrees. This put a damper on efforts to have the chameleon correctly imitate its real-world equivalent as it could not travel up any surface that was the slightest bit steeper than 180 degrees. Many trees had to have their meshes tweaked in order to better allow the chameleon to climb them. The character controller is instead a better solution for a more typical game where the ground is the intended surface for a character to move across.

For more advanced chameleon motion, a specialized character controller will have to be developed from scratch that will correctly cling to any surface it comes into contact with.

Handling the colour changes was the easiest script to implement: Unity's random number generator is good enough to ensure that every run through the game is very different in terms of what colours appear and for how long.

The problem of how to make the chameleon walk up trees was difficult to solve in that finding a solution that code could be modelled on proved difficult. This is mostly due to how most games have characters that walk upright across the ground, no matter how irregular. What the chameleon needed to do was somewhat obscure in that sense. Inheriting the ground normal was done easily enough, but keeping the forward direction proved difficult as just adopting the ground normal overrode the forward direction with the implicit forward direction from the ground normal, despite how there should have been none. The chameleon could correctly adjust itself to have all four feet on whatever surface it was walking across, but it would flip directions seemingly randomly as it was moved across the surface.

Overcoming this required using an obscure variation of the LookAt function along with some vector math that adjusted the forward vector to the ground. The LookAt function takes in the adjusted forward vector and the ground normal and correctly adjusts the rotation of the chameleon relative to the ground. His collider notably does not rotate with him.

Playing the correct animations and blending animations in and out is accomplished through a script written specifically for the chameleon. It checks the speed in order to determine whether to play the walk or idle animation, checks whether the chameleon's character controller collider is grounded (touching some surface) and if not plays the ground animation, and randomly plays a tail animation. If a rotation is applied, the correct turn animation is blended into the idle or walk cycle, as dictated by the speed the chameleon is moving at.

A particularly important part of the chameleon is the attached camera. The game relies heavily on visual elements, so the camera had to be flexible. A traditional over the shoulder camera would be limiting when the chameleon is in the trees, as would a first person camera, which would have the player looking out of the chameleon's eyes. Instead, an orbit camera scheme was implemented that allowed the camera to orbit around a

point a little ways over the chameleon, which allowed for the best view. Mouse motion controls the position of the camera.

3.5.2 SNAKE

The snake currently has its animation projected into the ground through a series of ray casting operations: every bone in the snake's spine projects a ray downwards to check how far away the ground is. If it finds no ground within a certain limit, it projects a ray upwards, just in case it is under the ground and needs to move up. This method mostly does not interfere with the animations, though it ruins the strike animation for the head and neck and does the same for idle. Further refinements will focus on ensuring that bones will point towards their parents so the spine will remain smooth.

3.5.3 ENVIRONMENT

A game object without an attached visual element currently coordinates the creation of the antidote in a random location. Several empty game objects, which only specify a location, rotation, and scale in world space, were placed around the environment. The script attached to these locations' parent object selects one of the locations at random and creates the antidote there, along with all the scripts necessary for detecting the chameleon's touch.

3.6 TESTING

This is done by having test players go through the game and report on any game elements they found annoying, any glitches they uncovered, and any other features that could use expansion. Final setting tweaking is done in this stage after figuring out how playable and beatable the game is.

3.7 SUMMARY OF TASKS

Task	Software	Time in Person Hours	Dates	Comments
Finding reference photos for chameleon	Google Chrome	4	Sept 20, 2010 - Jan 24, 2011	Difficult to find useful shots such as front, side, top, and bottom views.
Building chameleon mesh	Blender	22	Oct 22, 2010 - March 7, 2011	Had to redo some sections, the tail and the hands and feet, in order to get them into the proper rest position.
Rigging the chameleon mesh	Blender	25	Nov 7, 2010 - March 9, 2011	Had to redo rigging as well in order to properly redo the mesh. Took a long time because experimentation was necessary to find the best way to set up Inverse Kinematic constraints, as I had very little previous experience with them.
Animating the chameleon	Blender	32	Nov 20, 2010 - April 6, 2011	Took forever to develop a good walk cycle, with all the changes to the skeleton causing the loss of perfectly good key frames. Also, there was a steep learning curve when it came to figuring out how to make the Action Editor work.
Writing scripts for the chameleon	Unity/Visual Studio 2010	36+	Jan 29, 2011 - April 2, 2011	The hardest script to implement by far was the one that made the chameleon inherit the normal of whatever surface he was standing on while maintaining his forward direction. I spent weeks pouring over vector math guides, consulting forums and the documentation, and working out the math with my father. In the end, I stumbled upon the correct code almost by accident.
Texturing the chameleon	Blender	1	Oct 22, 2010 - Feb 26, 2011	Was pretty simple. Creating a separate vertex group for the eye kept getting put off in the face of bigger challenges.
Modelling the snake	Blender	2	Jan 10, 2011 - Jan 17, 2011	Was pretty simple, except that in the attempt to reduce the polygon count, more work fusing together extraneous vertices was made.
Rigging the snake	Blender	14	Jan 17, 2011 - Feb 27, 2011	This required a lot of research to get right as it proved that the initial plan to have a spline act as the bone would not work when imported into Unity. The rig was very detailed and complex and had more bones and constraints than the chameleon. It was worth it because manipulating the snake took way less effort during the animation phase.
Animating the snake	Blender	8	Feb 7, 2011 - March 17, 2011	A sinusoidal spline was built and used as a guide while building the slither animation in order to ensure that the curvature and speed was correct.
Writing scripts for the snake	Unity/Visual Studio 2010	12	March 18, 2011 - April 6, 2011	This is still a work in progress. Many avenues were investigated in an attempt to find the best way to project the snake's motion onto the

				irregular terrain, but in the end a crush projection method was used. It is still being defined. The AI has yet to be built.
Finding reference photos for the environment	Google Chrome	5	Oct 15, 2010 - Feb 16, 2011	Research was done into what areas the veiled chameleon lived in and what sort of climate would be acceptable and feasible to build.
Finding textures for the environment	Google Chrome/MS Paint/GIMP	7	Oct 15, 2010 - Jan 31, 2011	A good texture needs to be able to be tiled without looking weird. Finding natural textures like that is hard, so the best bet is to find many okay textures and mix them when they are applied.
Building the environment	Unity/Blender	7	Oct 15, 2010 - April 5, 2011	Building the irregular terrain took the least time, though much experimentation with different environment levels had to be done first. Building the trees was also a fairly short process. Texturing the environment took a while, as getting the textures to overlap correctly at the proper resolution took time.
Writing scripts for the environment	Unity/Visual Studio 2010	16	Feb 17, 2011 - March 25, 2011	Not many scripts were actually written for the environment, save for those that were applied to building the antidote. Mostly, the time spent here was on experimenting with getting the chameleon to walk on the trees.

4 GAMEPLAY

Currently, the chameleon is capable of climbing almost all elements in the game environment at the player's direction and the game ends at the end of the countdown with a loss for the player. The snake is no threat yet, nor does hitting the antidote, which is very hard to find, end the game in the player's favour.

Preliminary testing has suggested that "hints" might be necessary for the player in order to make the game feasible when it is possible to win upon touching the antidote. Possible solutions include having other fruits scattered around to be picked up that will, for a short time, give the player the chance to see the scent trail of the antidote in the form of a particle emission moving with a random wind.

5 CONCLUSIONS

Blender appears to be well on its way to becoming a 3D studio solution capable of competing with commercial solutions. While the user interface is dull and at times confusing, it seems that version 2.5 is making inroads on those two deficiencies. Blender, while lacking a lot of the custom solutions for specific tasks like handling atmospherics, has all the basic tools necessary to do it the hard way. A determined amateur would definitely be able to utilize Blender to create 3D elements for his game to his satisfaction with the help of the tutorials and manuals available online.

Unity is a very useful tool. The documentation is at times unclear though, making it difficult to understand what a lot of the built in functions do. There are many active forums devoted to Unity, along with plenty of resources, from tutorials to free script packages for specific tasks. Its resource requirements in terms of hardware are quite strenuous though: a higher end processor and video card are a must, or Unity will freeze or jump frames when running through gameplay.

Unity does have stability issues, as well, at least when run on a Windows 7 machine with 64 bit architecture while running ESET or Kaspersky antivirus. It will refuse to start up at times. However, the Unity team is looking into the issue and issues regular updates, so the problem may be resolved soon.

6 FUTURE WORK

Work yet to be done includes creating an artificial intelligence for the snake such that it can detect a moving chameleon of the wrong colour within a sphere defining its area of awareness and attack it successfully. The major challenges are in detecting when a colour is wrong—does not match its surroundings—and navigating to the chameleon over the bumpy terrain without going through the ground or some obstacle such as a tree or rock. The chameleon crouching must also be made to cause the snake's attack to fail. Snakes will be placed randomly eventually, some even in trees, once the AI is capable of navigating in three dimensions.

Also, screens for winning and losing must be created. Game logic enforcing the time limit currently calls the losing screen, but touching the antidote must be made to call the win screen.

Detecting terminal velocity on the chameleon has yet to be done. This will probably involve a velocity check alongside a “grounded” check. Failing both the conditions will cause the death animation to play upon touching the ground and the loss screen to be loaded soon after.

Deep water has yet to be defined for the chameleon. An invisible trigger will have to be defined for the irregular area that is “too deep” for the chameleon.

Also, the snake and the environment must be textured completely. The snake is currently lacking textures of any sort, while the environment is missing some rocks and trees.

7 REFERENCES

Blender download site: <http://www.blender.org/download/get-blender/>

Unity download site: <http://unity3d.com/unity/download/>

XNA download site: http://create.msdn.com/en-us/home/getting_started

XNA review: <http://www.fairyengine.com/articles/xna1stlook.htm>

Blender Hotkey Guide: http://en.wikibooks.org/wiki/Blender_3D:_HotKeys/All

Blender IK Tutorial:

http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/Advanced_Tutorials/Advanced_Animation/Guided_tour/Const/ik

Getting Started with XNA and Blender: <http://www.virtualrealm.com.au/Blog/tabid/62/EntryId/440/Getting-Started-with-Blender-3D-and-XNA.aspx>

Blender Texture Guide: http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/Using_Textures

Blender Game Engine Tutorial: <http://www.youtube.com/watch?v=SZNstSGcDVA>

Blender Weight Painting Tutorial: <http://www.youtube.com/watch?v=DWVzqorScTM&feature=related>

Getting Python Installed for Blender: <http://www.blender.org/forum/viewtopic.php?t=11938>

Blender Leg Rig Reference: http://wiki.blender.org/index.php/Doc:Tutorials/Animation/Armatures/BSoD/Leg_Rigs

Blender Constraint Guide: http://wiki.blender.org/index.php/Doc:Manual/Constraints/Common_Interface

Blender IK Solver Guide: http://wiki.blender.org/index.php/Doc:Manual/Constraints/Tracking/IK_Solver

Blender Stretch To Constraint Guide:

http://wiki.blender.org/index.php/Doc:Manual/Constraints/Tracking/Stretch_To

Blender Floor Constraint Guide: <http://wiki.blender.org/index.php/Doc:Manual/Constraints/Relationship/Floor>

Blender Spine Rig Guide: http://wiki.blender.org/index.php/Doc:Tutorials/Animation/Armatures/BSoD/Spine_Rigs

Blender Linear Curve Tracking Rig Guide (this was used for the snake):

http://wiki.blender.org/index.php/Doc:Tutorials/Animation/Armatures/BSoD/The_Linear_Curve_Tracking_Spine

Foliage reference: <http://woophy.com/images/photos/859/3/l/101501.jpg>

Blender Material Guide: http://wiki.blender.org/index.php/Doc:Tutorials/Materials/BSoD/exercise_2

Blender Tutorial List: http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/Tutorial_Links_List

Blender IPO Editing Guide: <http://wiki.blender.org/index.php/Doc:Manual/Animation/Editors/Ipo/Editing>

Blender Action Editor Guide: <http://wiki.blender.org/index.php/Doc:Reference/Windows/Action>

Blender Array Modifier Guide: <http://wiki.blender.org/index.php/Doc:Manual/Modifiers/Objects/Array>

Unity Introductory Tutorial Set: http://www.3dbuzz.com/vbforum/sv_showvideo.php?v=3611

Chameleon Walk Cycle Reference Video (link to be terminated as of April 29th):

<http://video.google.com/videoplay?docid=-6943357287200570280#docid=3552395660062478973>

Unity Debugger Guide: <http://unity3d.com/support/documentation/Manual/Debugger.html>

Unity iTween Plugin Reference: <http://itween.pixelpacement.com/documentation.php>

Unity Fading Objects Between Camera and Target Post: <http://forum.unity3d.com/threads/69955-How-to-fade-objects-between-camera-and-character?highlight=sight>

Blender [Modeling a Simple Person](#)

Blender [Quickie Material](#) and [Multiple Materials Per Object](#)

Blender [Bones](#)

Blender [Creation and Controls](#)

Blender [An aMAZEing game engine tutorial](#)

Blender [Platformer: Physics Fixes](#)

Blender [Platformer: Physics Fixes](#)

Blender [Build a skybox](#)

Blender [Basic mouse pointer](#)

Blender [Text in BGE](#)

Blender [Python Platformer: Creation](#)

Blender [Armatures](#)

XNA Tutorial 1: Displaying a 3D Model on the Screen

XNA Tutorial 2: Making Your Model Move Using Input

XNA Optional Step: Controlling the Ship with Keyboard Input

XNA Tutorial 3: Making Sounds with XNA Game Studio

XNA Tutorial 4: Make a Game in 60 Minutes

XNA FuelCell: "Ships" Passing in the Night (series of 6 tutorials)

XNA How To: Render a Model

Unity LookAt: <http://unity3d.com/support/documentation/ScriptReference/Transform.LookAt.html>

Unity General Script Reference: <http://unity3d.com/support/documentation/ScriptReference/>