Abstract

The character has a great influence on the game play with help of the user. The user acts as the character when visiting the world that the game displays. This makes the character very important for both the user and the game.

In this report different techniques on how to create a character are explained, which includes modeling the character, creating a skeleton for the character and animating the character. These parts are needed to create a character for a game.

A plug-in for the game engine Crystal Space is created. The plug-in that is implemented uses a character file format to load a character and manage it. The file format is an external file that can be created with help from different tools. The plug-in manages the meshes, the skeleton, the textures and the animations of the character. With help of the plug-in, it's possible to add a character into a Crystal Space project.

There are several character file format and some of them is examined in detail. The best-suited character format is used for the Crystal Space plug-in.

A demo application that uses the plug-in for Crystal Space is made to show the features of the plug-in. The demo application has user interaction through the keyboard and a camera management, which makes sure that the character is always visible through the camera view. The camera management takes care of the interaction of the camera and the surroundings so that the camera doesn't prevent the character from moving. The demo application is also used for testing purpose, to check if the plug-in is working correctly.

There are different techniques on how to manage a camera. Since the character is the most important part, the camera uses a third-person view. The techniques for a third-person view are described. When the camera collides with the surroundings, a new position for the camera is calculated so that the camera isn't located inside an object or outside its sector.

Contents

Chapter 1

Introduction

The character has an important role in a game. The user takes control of the playable character and interacts with the world it has been put into. The input user makes to control the character affects the game scenario. The character is for the most time visible to the user during game play, making the design of the character very important. How the character looks like and acts have a great effect on the user.

Since the character have a huge influence on the user and the game play, it's important to create the character correctly right from the start. The design, the clothes and the movement of the character are all important when creating the character. The animations have to be realistic and correctly implemented.

Creating a standard for a character management makes the interaction of the project much better. With a good standard, it's possible to reuse textures, skeletons and animations on different characters, making character development much easier. A good character management isn't dependent on the game itself, which makes it possible to use in other games. Using the same standard in different games makes it possible to combine characters. It also makes the development time for the project shorter since the character management is already finished and don't have to be recreated.

1.1 Purpose

The project is about humanoid models that are used in games. In particular it will be based on the open source game engine Crystal Space.

The development of the game engine is getting bigger all the time and more and more people are using Crystal Space to improve it or use it for their own project. Even if Crystal Space is a large game engine with several features, there are things that are missing or not so good as it could be. This is mainly because Crystal Space is still in beta stage, and no final release has been done. One thing that is missing in Crystal Space is a character implementation.

The goal of this project is to implement an extension to Crystal Space that supports humanoid models. The humanoid models can be used as characters in games, but also bots if behavior is added to the model. The models will support appearance and motion. The animations of the model shall be as realistic as possible.

Hopefully the character implementation will help other developers that are working with Crystal Space to easily add characters to their project. Since the project will be open source, that is the source is available for the public, it is possible to convert and port the character implementation to other game engines as well.

1.2 Overview

This report consists of two main sections, an analysis and an implementation description. In the analysis section, a study in different areas is made. A study on different methods and formats in modeling and animating a character is made to determine what method that shall be used in the project. A demonstration is made to debug the character implementation and to demonstrate the possibility of the implementation. A study in different techniques that can be used to manage a third-person camera view is made to determine how the camera management will be implemented. Also the mathematical calculations of the camera management are described in detail. A conclusion of the analysis is done at the end of the analysis section to get a view on how the implementation shall be made.

An implementation of the format that shall be used in the project is described in the implementation section. The implementation of the character support is then made as a plug-in for Crystal Space. The plug-in loads and manages a character so it can be viewed and used in a Crystal Space project. A demonstration application that is used to test and show the features of the plug-in is also described in detail in this section.

1.3 Crystal Space

Crystal Space is a free 3D Game Development Kit, which can be used for a variety of 3D visualization tasks. Crystal Space is developed in C++ and it's a large open source project that falls under the GNU copyright license. It runs on various platforms such as Linux (Unix), Windows, MacOS/X and BeOS. Crystal Space supports various libraries and plug-ins.

Crystal Space is actually a package of components and libraries. Most of the components and libraries are intended for a 3D environment. But some of them could also be used for 2D environment such as sound drives and networking components. The components and libraries in Crystal Space are independent of each other. That means that it is not necessary to include the sound drives in the project if they are not needed. They can be used on their own. Even if the components and libraries are independent of each other, there is a high level of integration. The components and libraries are designed so that they can be used together.

The Shared Class Facility (SCF) system is the core low-level memory framework for Crystal Space. The SCF divides Crystal Space into modules that communicate with each other. In early development of Crystal Space, COM (Component Object Model) was used instead of SCF. SCF was implemented with COM as a testbed and because of that, SCF is somewhat similar to COM. The object registry serves as a registry for all objects in Crystal Space. The objects used in Crystal Space will be registered to the engine through this registry. Everything in Crystal Space uses the object registry to get the pointer to the wanted object. The plug-in manager takes care of all shared libraries (plug-ins) used in Crystal Space. It is responsible for loading and unloading the plug-ins. The Crystal Space framework itself is a plug-in so the plug-in manager is important in Crystal Space.

Unlike libraries, plug-ins can only provide SCF interfaces. A library can provide classes and optionally also SCF interfaces. A plug-in can be organized as static library or dynamic linked library (DLL). To be able to use a plug-in in a Crystal Space application, the library that contains the plug-in first have to be registered into the Crystal Space configuration file. Then the plug-in have to be loaded in the application. To access the

functions of the plug-in, the plug-in SCF interface has to be queried within the application by using the statement CS_QUERY_REGISTRY().

Crystal Space has something called the Virtual File System (VFS). A problem that occurs when using different operating systems is the file system. The file system is different on different operating systems. The VFS solves this problem by using an own file system that works on all platforms. Some extra features in the VFS include archive support and overlaid directories. The VFS is constructed with a configuration file. The configuration file can easily be edited if a new directory needs to be added to the VFS.

Crystal Space uses so called map files in order to store a world or an object on an external file. It is a script file that can be edited from any text editor. The map file contains a description on how the world will look like. Objects, materials, plug-ins and different other things that is needed to build up a world is described in the map file. A Crystal Space world could also be implemented directly into the application.

Crystal Space uses sectors to build up a world. A world in Crystal Space built with one or more sectors. A sector is basically a region of space. The sector can contain several mesh objects such as walls. To connect the sectors with each other, portals can be used. The portal will display the sector that the portal points to.

Chapter 2

Character Modeling

The description of the character modeling is divided into two parts. The first parts are a study in different techniques to model and animate a character, and have a detail description on how the characters can be created. The anatomy and skeleton of the character is examined and described on how to create them. Also different animation techniques and how to move the character is described. This information is needed to determine an appropriate file format to be used in the character plug-in for Crystal Space.

The second part is a study in finished file formats that can be used for character models. The file formats that are examined in detail are the Crystal Space mesh object, the H-Anim Standard and the Cal3D library. From these file formats, the best-suited format for the character plug-in is chosen.

2.1 Character Creation

Creating a character for a game can be done in three steps. These steps are modeling, animation and behavior. Modeling includes both the skin and the skeleton of the character. The skeleton isn't seen during game play, but is used for animation. The skeleton can also be used as a template for creating the skin for the character. The animation for the character is used to make the character interact with the surroundings and to make the player interact with the character, for example when the character moves or being hit by an enemy. The behavior of the character is how the game makes the character interact with the surroundings. When the player interacts with the character, behavior is usually not used. The behavior is also known as AI (Artificial Intelligence). Since the behavior of the character will not be a part of the project, it will not be described here.

2.1.1 Modeling

Anatomy

In a 3D world, the characters are constructed with something called meshes. The whole character body is created with meshes. Usually when modeling a character the meshes are created from a template. It is also possible to model a character without a template and create it on free hand, but then there is a big chances that the proportions of the character can become unrealistic. A template that is commonly used when

modeling a 3D character is to use two images of the character. The images are drawn by hand or can even be a photo of someone. One image should be the front view of the character and the other image should be the side view of the character. Usually it is enough to use the front and side view of the character to create a 3D model, but a back view and a top view can also be used if necessary. Another template can be a skeleton of the character. The skeleton shows the length and position of each body part so the proportions becomes right from the beginning when modeling. A third way to use a template is to edit a finished character, for example changing the face, clothes and maybe the height of the finished character so it becomes a different character, but still having the right proportions.

There is much to learn from looking on the human body when creating a character or creating a template for a character. But the human body is a large and complex hierarchy. One change on the hierarchy can change the movement drastic. To animate the character correctly, all parts must be in right place in relation to the rest of the body. The human anatomy is the perfect blue print to accomplish that. It will be helpful when creating a realistic character.

When modeling the character, it's common to start to create the head or the trunk of the character, because these are the significant parts of the body. The head is probably the most different body part on each character and the trunk is the center of the character, so it is common to start on one of these body parts when modeling. After that, the arms and legs are created in no particular order. A technique that can be used when modeling a character, which speeds up the modeling process a little, is to only create a half character. For example when creating the head, only the left side of the head can be created. When the left side of the head is finished, copying and mirroring the left side can create the right side. Of course, on a real human, the left and right sides of the face is different from each other, but this is not important in games.

Skeleton

The main purpose of building a skeleton for the character is to create an easier animation. The skeleton is attached to the meshes of the character. This way, when moving a skeleton joint the meshes automatically follows the movement. When creating the skeleton model you basically are creating a bunch of joints and connect them to each other. These joints are similar to the movable joints on the human skeleton such as elbow, knee, wrist and so on. When moving one joint, it will affect on the nearby joints as well. For example, if you move up your elbow, your wrist is also moved. There are different kinds of joints on the human body that moves differently. For example the hip moves differently than the knee. The hip has a ball and socket joint, which have rounded surface, allowing greater freedom of movement. While the knee have a hinge joint which only can be moved forward and backwards. Usually, these features aren't included in the skeleton model.

To get a realistic animation, the placements of the joints have to be in right position in relative to the body mass. Problems occur when the skeleton is placed in the middle of each part of the character. For example, the spine of the skeleton is in the middle of the torso. But looking at how the human skeleton is build within the body, you see that this isn't the case. The placement and construction of the spine in the human body makes it easy to bend forward, but hard to bend backwards. This creates limits of the human movement. These movement limits lies within the skeleton and can be implemented by adding limit attributes for each joint on the skeleton model. But this is not common because the animation is based on data that is read in the program and the character moves according to the data.

Stefan Henry-Biskup explains the problem with Jack Nicholas 5 in his article where the golfer's posture was very stiff and unnatural, the shoulders were ballooned out, and the thighs looked too long (Henry-Biskup 1998). To fix the problem with Jack Nicholas 5 the developers tweaked the vertex attachments, applied bulge angles, and editing link parameters to fix the animation on the golfer. That took a lot of time and work. A better

way to avoid the problem with Jack Nicholas 5 is to create a good and steady skeleton before starting to model the character.

Looking at the movable joints of the human body it is possible to see where exactly the bending takes place. For example, if the shoulder joint is located in the middle of the shoulder mass, the shoulder will be ballooned out just like the Jack Nicholas 5 problem. To avoid the problem, the placement of the shoulder joint should be close to top of the shoulder mass. Stefan Henry-Biskup explains the placements of some joints to make a realistic animation in his article (Henry-Biskup 1998). One bone joint in one part of the body behaves similarly in other locations. Example the ankle is similar to the elbow and the joints of the finger and toe bones behave just like the knee.

A common mistake is to create the meshes of the character before the skeleton. When animating the character the placement of the body parts looks awful and not natural at all. Its possible to fix the placement of the body parts by changing the vertices and the links, but that takes a lot more time and effort. Before creating the meshes for skin and clothes Henry-Biskup recommends to create a skeleton for the character first (Henry-Biskup 1998). Build the skeleton before you build the mesh so that you can build the model's surface by aligning your geometry to the appropriate bones of the skeleton as you go. The skeleton is the base of the character when animating it.

2.1.2 Animation

Techniques

There are several techniques to use when animating a character. Each of them has their own advantages and disadvantages. Four common animation techniques are performance animation, keyframing, inverse kinematics and dynamics based animation.

Performance Animation
Performance animation, or motion capture is a technique where real-life movement is recorded for each frame of the animation. One way to record the motion is to map the motions from a real human being and export it to data in a computer. The advantage of this is that the data is based of real movements and therefore be realistic. But the disadvantage of this technique is that the motion has to be recorded again for each new animation. An alternative to performance animation is keyframing.

Keyframing
In keyframing, the animation consists of automatic generated frames called in-betweens. There are two kinds of keyframing, which is image-based keyframe animation and parametric keyframe animation. Interpolating the keyframe images creates the in-betweens in image-based keyframe animation. In parametric keyframe animation the animation is based on an entity such as object, camera or light and its parameters. The animation occurs when changing the values in the parameters at a given time. Keyframing has some disadvantages such as a lack of smoothness in motion. Other disadvantages are interrupts in the speed of motion and distortions in rotations.

Inverse Kinematics
Kinematics is the science of motion without regard to forces. Inverse kinematics is a specification of end point positions. The angles for each joint in a skeleton are then automatically determined. The disadvantage with inverse kinematics is that it doesn't take physical facts such as gravity and inertia into consideration, which makes animations more realistic. But the advantage with inverse kinematics is that the animation can be used on several different characters. It isn't necessary to record the motion again for each new animation like performance animation.

Dynamics Based Animation
Basic physical facts are used in dynamics based animations like gravity and inertia. In dynamics based animations the movements is a reaction of forces and torques. The main goal of computer animation is to synthesize any desired effects evolving through time: motion (position and orientation changes) and deformation (shape changes) (Thalmann 1996). The advantages with dynamics based animation is that the animations will be realistic due to the use of physical laws and for a given physical object, a physical model represents the class of all possible movements.

Movement

To create a realistic animation for the character, understanding the movement of the human body is necessary. JM Rinquet presents three rules when he talks about animation in a 3D world in his article. Gravity, and how it is used to create movement, is the first rule of three-axis animation and the two other rules are balance, tilt and twist (Rinquet 2001). This is easiest to perform with dynamics based animation because this technique uses physical facts. The center of gravity on a human is located between the hips. The limbs have to change the location of the center of gravity to be able to move the body. Every movement is either created by gravity or is a reaction of gravity (Rinquet 2001). Also, every movement of the body is based on a thrust, which result in moving the center of gravity and putting the character out of balance. The movement ends by regaining balance.

JM Rinquet uses walking as an example when showing the usage of gravity, balance and tilt and twist (Rinquet 2001). In his article he describes that the character is in balance at first. At the point where the character lifts the right foot, the center of gravity is displaced. The whole body is moving forward and the center of gravity passes the front of the supporting left leg, which makes the character out of balance, and the right foot touches the ground to regain balance. During that movement, the whole body reacts. The character tilts the hipbone to the left and tilts the shoulder to the right when lifting right foot. The entire body reacts on one movement to keep it in balance.

2.2 Finished Character Models

There are three different character file formats that are examined. One of these file formats will be used for the project. The formats are the Crystal Space mesh object, the H-Anim Standard and the Cal3D library. Each of these formats is described in detail and the features of each format are brought up.

The Crystal Space mesh object is examined because it is a part of the Crystal Space engine. The file format interacts with other Crystal Space components, making this format easy to use in a Crystal Space project. The H-Anim (Humanoid Animation) Standard is a description on how a character is designed. Since this format is a standard, it makes easier to use and convert other characters using this standard, making it a more widely used file format. The Cal3D library is examined because it's designed to be used for games and it's platform independent. It has several unique features that make character creation easier.

2.2.1 Crystal Space Mesh Objects

The mesh object plug-in system is a general system for defining various kinds of 3D objects to be used in the Crystal Space engine. With help of the mesh object plug-in system, it's possible to define your own 3D object that can be used in a Crystal Space project. There are several predefined mesh objects in Crystal Space already such as a ball, 2D sprites and particle systems.

Each mesh object knows how to render and light itself. To use a mesh object in Crystal Space, the mesh object should be registered to the engine so that the engine can take control over the object. The engine is responsible for the position of the object in the

engine. The engine is also responsible for deciding when to draw and update the objects. Each mesh object implements the interface *iMeshObject*. Which is an interface that the engine recognizes and uses to control the mesh object. But the mesh object could also use other interfaces that the engine doesn't know about. For example, many mesh objects uses besides the iMeshObject interface, also the *iPolygonMesh* interface, which is used for collision detection on the object. To check if a mesh object uses a specific interface, the statement SCF_QUERY_INTERFACE() could be used.

As a shell to the mesh objects, a Mesh Object Factory is created. Mesh object instances can be generated from the mesh object factories. The mesh objects generated from the mesh object factory inherit some of the characteristics from the factory. Which characteristic that is inherited depends on the Mesh Object Type, which describes a specific kind of mesh object plug-in. For example types such as cube, ball or snow particle system. The factories could be used for different possibilities. One possibility could be that the factory is empty and only serves for creating mesh object instances. Only one factory could be used for this possibility. Another possibility would be that the factory contains default values that are copied to all mesh objects that is created from this factory. The characteristics for the factory could be changed before the creation of an instance to get different characteristics on the mesh object instance than the default ones. A third possibility would be to use same characteristics on all mesh objects created from that factory. Changing the characteristics on the factory will change the characteristics for all mesh objects created from that factory. Which possibility that is used depends on the mesh object type. The mesh object interfaces uses the *iMeshObjectFactory* interface. Just like the mesh objects, the factories could use other interfaces besides the iMeshObjectFactory interface.

As mentioned above the engine is responsible for management of the position of the object and when to drawn and update it. The engine uses the iMeshWrapper interface to handle these responsibilities. The iMeshWrapper holds the references to the iMeshObject instances. There is two ways to use the mesh objects in Crystal Space. One way is to code it directly into the source code of the application and the other way is to use the map file format.

Using Map Files

Using the Crystal Space's map file is the easiest way to use mesh objects. The first thing that is needed for using the mesh object is to specify an entry directly inside the WORLD statement. Here is an example that uses the fountain plug-in:

```
...

MESHFACT 'fountainFactory' (
  PLUGIN ('crystalspace.mesh.loader.factory.fountain')
  PARAMS ()
)

...
```

This example uses the mesh object loader plug-in by the PLUGIN() statement to load and register a factory to the engine. The PARAMS() statement is empty because the characteristics aren’t defined here. Because of that, only one factory is created.

To place the mesh object in the scene the MESHOBJ statement is used. Here is an example for placing the fountain object on the scene:

```
...

MESHOBJ 'fountain' (
  PLUGIN ('crystalspace.mesh.loader.fountain')
  PARAMS (
    FACTORY ('fountainFactory')
    NUMBER (300)
    MATERIAL ('spark')
    ORIGIN (0,0,0)
```

```
    DROPSIZE (.05,.05)
    COLOR (0.7,0.9,1.0)
    ACCEL (0,-.1,0)
    FALLTIME (3)
    SPEED (1)
    ELEVATION (1.5)
    AZIMUTH (0)
    OPENING (.2)
    MIXMODE (ADD ())
  )
  MOVE (V (-10,-1,14) MATRIX (ROT_X (1.5)))
)

...
```

This example uses the loader plug-in to register the object in the engine. The example defines the characteristics that are used for the object in the PARAMS() statement such as color, material etc. The MOVE() statement defines the position of the object and is outside the PARAMS() statement because the engine and not the object itself control the position of the object.

Using Directly in Code

When using the mesh object directly from the code, some steps have to be performed. Here is example code for using the ball plug-in:

```
...

// Get the ball mesh object plug-in.
ball_type = CS_LOAD_PLUGIN (this,
  "crystalspace.mesh.object.ball", iMeshObjectType);
if (!ball_type)
  Printf (CS_MSG_WARNING, "No ball type plug-in found!\n");
ball_factory = ball_type->NewFactory ();
```

First, the mesh object has to be loaded by using the statement CS_LOAD_PLUGIN(). Then at least one mesh object factory has to be created by calling NewFactory function on the mesh object instance.

```
// Make a ball using the ball plug-in.
iMeshObject* ballMesh = ball_factory->NewInstance ();
iBallState* ballState = SCF_QUERY_INTERFACE (ballMesh, iBallState);
ballState->SetRadius (.5, .5, .5);
ballState->SetMaterialWrapper (material);
ballState->SetRimVertices (12);
```

Calling the function NewInstance on the factory will create a mesh object. The characteristics of the mesh object can be defined by using a statement called SCF_QUERY_INTERFACE() to receive a handle to the interface that can be used to change or set the characteristics. The statement can be used in both mesh object factory and mesh object if they have characteristics that can be changed.

```
iMeshWrapper* ball = engine->CreateMeshWrapper (ballMesh, "MyBall", room);
ballMesh->DecRef ();
ball->GetMovable ().SetPosition (csVector3 (-3, 5, 3));
ball->GetMovable ().UpdateMove ();
ball->DeferUpdateLighting (CS_NLIGHT_STATIC | CS_NLIGHT_DYNAMIC, 10);

...
```

Each mesh object instance needs a mesh wrapper instance. With the mesh wrapper, the position and transformation of the object can be set. This example loads the ball plug-in and places it at the position (-3, 5, 3) in the 'room' sector and then it updates the 10 closest lights.

2.2.2 H-Anim Standard

The Humanoid Animation Working Group or H-Anim for short is working on a project for standard design for creating interchangeable humanoids using VRML. VRML stands for Virtual Reality Modeling Language and is a file format for describing 3D scenes and objects. VRML is used for displaying 3D graphics in interactive virtual worlds on the web. The latest version of the H-Anim standard is H-Anim 2001, which are using VRML97. The project was formed so that developers could use a standard naming convention for humanoids. The standard will help developers to easily create humanoids by reusing code or animations so that the development time will be less.

Construction

Several nodes define the H-Anim file. There are five different nodes in the file. The nodes are: *humanoid*, *joint*, *segment*, *site* and *displacer*. The joint, segment, site and displacer nodes defines the humanoid while the humanoid node is a form of header for the H-Anim file. The humanoid node contains data about the humanoid such as author and copyright information. There are also references to all the joint, segment and site nodes in the humanoid node.

The H-Anim file contains a set of *joint* nodes. A joint on the human body is for example an elbow or a wrist. Each joint node can contain other joint nodes. The joint nodes describe the articulations of the humanoid, which is organized into a hierarchy that describes the skeleton of the humanoid. The application that uses the H-Anim file can retrieve information such as upper and lower joint limits, the orientation of the joint limits and a stiffness/resistance value from each joint node. This information can be used in inverse kinematics.

Each joint node can contain a *segment* node or other joint nodes. A segment node represents each body part of the humanoid, for example hand or foot. The segment node can only be defined as a child of a joint node. The application that uses the H-Anim file can retrieve information such as segment mass from each segment.

Each segment can have a number of *site* nodes. The site nodes define the location relative to the segment. The site nodes can be used for attaching clothing and jewelry. The sites can also be used for inverse kinematics and to define eye points and viewpoints locations.

A displacer node can be used in three different ways. It can be used to identify the vertices on a segment. It can also be used to represent a particular muscular action, which displaces the vertices in various directions. The third way it can be used as is to represent a complete configuration of the vertices in a segment. There could be for example a displacer for each facial expression on a face. A displacer is often use to control the shape of the face, but they could also be used for other body parts, for example changing the shape of an arm segment as the arm flexes.

Animation

The H-Anim file format also includes animations such as limb movement and facial expressions. With help of the standard, it will be easy to use one animation on several other humanoids. The H-Anim humanoids can be animated using keyframing, inverse kinematics, performance animation systems and other techniques.

Coding

The file format makes it easy to write the code by hand, but can be time consuming the more details the humanoid have. There are authoring tools that support the H-Anim format that can be used to create H-Anim humanoids. The tools make it easy to create humanoids. Some tools that supports the H-Anim standard is X3D-Edit from Don

Brutzman, Spazz3D from Virtock Technologies, SkeletonBuilder from Tpresence, Vorlon from Trapezium and Avatar Den from blaxxun.

2.2.3 Cal3D Library

Cal3D is a 3D character animation library written in C++. The library was originally designed for a 3D client for Worldforge. The Cal3D project is open source and can be used for many different projects and platforms. Cal3D falls under the GNU Lesser General Public License.

The core feature of the Cal3D library is that it was built up as skeletal-based. The characters that supported Cal3D were built up with a skeleton so it gives a lot of freedom over the animation process. Besides the skeleton, there is also a control system that handles the animation sequence and animation blending. The handling of the meshes and materials in Cal3D would make it possible to completely change the look of a model. Some advanced features are included in Cal3D such as level-of-detail while other features such as collision-detection and physics isn't included yet, but might be in the future releases.

The developers of the library avoided making it platform dependent. The library must be portable so the library was coded in C++ and only depends on the STL (Standard Template Library). STL is a library that contains classes and common algorithms that handles common programming tasks. STL is portable on various operating systems. Also the performance was very important when Cal3D was created, because it was originally developed for online games where performance is an important issue. So the library was optimized.

The Classes

Cal3D has a set of Core Classes. The core classes represent one model type and stores all shared data for that model type. The Instance Classes represents one specific instance of the model type and are constructed from the core classes. That makes it possible to have one specific type of character but creating several characters from that type. That doesn't mean that all the characters created from the core class have to appear the same way. The core classes contain all the animations, materials and meshes for that model type. But the instances could use different materials and that way, appear in a different way even if the instances comes from the same core class.

The core class contains four parts of data for one model type, which is:

The hierarchical structure such as skeletons and bones
The motion data such as animations, tracks and keyframes
The surface properties such as materials
The body parts such as meshes and submeshes

The instance class contains three parts of data that is used for each instance of a model type:

The current state of the skeleton
The active set of animations
The attached body parts

Besides the three data parts, the instance class also has helper classes that simplify the model handling such as motion control and rendering interface.

Construction

The Cal3D library builds up the character with one or more meshes. Each mesh has a number of triangular faces. Each face is assigned to a corresponding submesh, which

holds all faces with the same material. Each submesh contains lists of vertices and faces.

The animation of the Cal3D character is skeletal-based so all the meshes are attached to one or more bones. The bones create a skeleton structure. To animate the character, only the bones need to be adjusted so that the pose of the skeleton is changed. This makes it easy to animate the character. Each animation sequence is stored separately in the core model. This way, you can use the same animation on other characters. Each animation sequence stores the track for each bone that is affected by the animation. The other bones aren't affected by the animation. That way, blending two animations becomes clearer.

Cal3D uses an animation control system called the Mixer. The Mixer handles the animations on the character. Each character has an active animation set. This animation set is holds the current animation that the character is performing. The animation set could have one or more animations. That way, a character could have for example two animations running on the same time. If one animation would be a walking sequence and the other a waving sequence, running those two will make the character walking and waving at the same time. The Mixer is responsible for updating the active animation set. The Mixer has also support for fading out and fading in between two animations. Also, each active animation has a weight associated with it. This means that the Mixer controls the amount of influence that the active animation has on the character.

The core model also stores the material that will be used on the character. The materials include the textures, the shininess factor and the color components such as ambient, diffuse and specular color. Every submesh has a material assigned to it. It is possible to change the material that is assigned to another material. This way you can for example make the character change clothes by only changing the materials that the character is using.

There is also something called the Renderer in Cal3D. The Cal3D library doesn't handle the actual rendering since that will make it platform dependent, but provides an interface to access all needed data for the rendering. The Renderer keeps data so it will be easy to access when rendering such as material data, deformed vertex data and face data among others.

The Cal3D library supports Level-of-Detail (LOD), which means that the number of triangles that will be rendered on the screen can be reduced. Reducing the LOD will increase the performance and can be useful when running on slow computers.

The File Format

The Cal3D file format is actually several external files. There are six file types that the Cal3D format is using. The main file type is the configuration file and has the extension .CFG. The configuration file contains the path name to all external files that the character are using from these file types:

The skeleton data (.CSF)
The animation data (.CAF)
The mesh data (.CMF)
The material data (.CRF)

There is only one skeleton data file for each character, but there can be several animation, mesh and material data files for one character. The last file type that the Cal3D format is using is the raw data of the textures that the material data uses. The raw data shall not be defined in the configuration file, but is already defined within the material data itself. Only characters that use textures have this file format. The extension of the texture data is .RAW. Here is an example of a configuration file for a character called Paladin:

```
#
# model: paladin
#

scale=1.0

skeleton=paladin.csf

animation=paladin_idle.caf
animation=paladin_walk.caf

mesh=paladin_body.cmf
mesh=paladin_cape.cmf
mesh=paladin_loincloth.cmf

material=paladin_head.crf
material=paladin_chest.crf
material=paladin_flesh.crf
material=paladin_shoe.crf
material=paladin_ponytail.crf
```

The number character (#) indicates that the line is a comment and should be skipped when the configuration file is read. The scale command indicates that the character will be scaled to a size that is two times bigger (in this example) than its original size.

Today the file format used for Cal3D is not well documented and doesn't have many tools that support the format. The only tools that support the Cal3D file format are 3D Studio MAX exporter and MilkShape exporter.

2.3 Summary

Using the Crystal Space mesh factory specification have some disadvantages. For example each part of the character can only assign one material to it and it will always use same material. This isn't a very critical disadvantage, but the possibility to change material during runtime can be very useful at game development for example when a character change clothes or appearance. But a more obvious disadvantage is that the animation sequence is stored frame by frame. Two animations can't be combined or run simultaneously. For example if the character have two animation sequences which are running and waving, those two can't be combined during runtime, but have to use a new animation sequence in order to get a similar effect. Also, animation sequences can't be reused for other characters. That way, each character must have its own animation sequence.

The H-Anim format is based on VRML coding so it can be used in a VRML scene. To be able to use the H-Anim format, the VRML coding have to be converted into a Crystal Space environment. This would be a hard task since VRML is a huge language with several features. It's possible to make the implementation easier by only implementing the basic of VRML, but that would limit the different ways to create H-Anim characters and the compability of other H-Anim characters. The H-Anim format is also a very complex hierarchy that includes almost every joint on the human body. A skeleton that detailed is overdone and not necessary when developing a character for a game. It will also lower the performance of the game during runtime. A detailed skeleton can be useful in future games where the performance have been increased, but today it's better to avoid it in order to increase the frame rate. It's possible to use the standard for itself and create an own file format for Crystal Space that looks like the H-Anim forma. That way the skeleton don't have to be so detailed and only include the necessary joints of the human body, but the problem with the limit on how to create a H-Anim character still remains.

The Cal3D library has an interface that is easy to use and understand which makes the implementation much easier. There are also several features included into the library that makes the library very interesting. Since Cal3D was created mainly for gaming, it would be an advantage to use it in the project. Each character has meshes, textures, a

skeleton and animations, which can be used for one or more characters. It's also possible to change those during runtime so it would be possible for example a character to change clothes by only changing a certain material of the character. The animations can also be combined and run simultaneously so creating animations will be easier. The Cal3D library is still during development and is getting better and bigger for each new release. In order to keep the high level of integration that Crystal Space has, the implementation of Cal3D is best done with Crystal Space's own mesh object format. That way, it will be easy to use the implementation in different Crystal Space projects and it will keep the compability with other plug-ins and applications within the Crystal Space environment. The mesh object is very common in Crystal Space.

Chapter 3

Camera Management

The camera will make the character in an application visible for the user. To see the complete character, a third-person camera view is used. But when the character moves, it will move away from the camera. A camera management has to be used to fix this, which makes the camera follow the character so that the character is visible all the time.

Over-the-shoulder view, which will be used in the camera management, is when the camera points right over the character's right shoulder. The camera is positioned behind the character a little bit above the character's head. The camera is always pointed on the same spot on the character regardless of the camera's position. The camera management that will be used will be done so that the camera won't prevent the character from moving.

This chapter is divided into three main sections, which are techniques, Crystal Space sectors and mathematic calculations. There is different ways to manage a third-person camera, which will be examined and explained in the techniques section. Advantages and disadvantages on each of the techniques are described, and the best-suited technique is chosen. In the section about Crystal Space sectors, an explanation on how Crystal Space uses sectors and how the camera management interacts with them is described. After that, the mathematic calculations that will be used for the camera management is calculated and explained how it works.

3.1 Techniques

There are two main parts on the camera management that will be used. These parts are the distance between the camera and the character, and the angle between the floor and the camera. These parts can either be fixed or flexible depending on the technique that will be used, which will give different effects on the camera view.

The techniques that will be examined are fixed distance and angle, fixed distance, flexible distance and a combination of fixed and flexible distance. The advantages and disadvantages for the techniques are explained.

3.1.1 Fixed Distance and Angle

In this camera management only two objects is needed, the character and the camera. The camera will be in a fixed distance behind the character and positioned above the character's head. The camera will have a fixed angle so that it will be pointed to the character and not straight forward. This way the character will be visible in the camera view. The camera distance and camera angle will always be the same so that the character will always be seen from a certain perspective.

The obvious problem here is that the camera will go through walls when the character is moving near the walls. This creates a big problem when the character passes a portal and the camera is above the portal since the camera is located above the character's head. To change sectors in Crystal Space, an object have to go through a portal. In this case, the character will change sector since it passes a portal, but the camera will go through the wall and not pass a portal. So the camera will still be in the old sector and not displaying anything in the new sector. A solution to this problem could be to include collision detection on the camera. This way, the camera can't go through the walls. In this case, the camera will stop the character from moving when a collision is detected on the camera. The camera will prevent the character from going through portals or openings that are located below the camera, for example a door opening.

3.1.2 Fixed Distance

Three objects will be affected with this camera management. The objects will be the character, the camera and the camera axis. The camera axis is a virtual line between the character and the camera, which is not visible. The collision detection will be checked both in character and the camera axis. The camera doesn't need any collision detection because it will always be attached at the end of the camera axis and the camera axis handle the collision detection for the camera. The length of the camera axis is the distance between the character and the camera. The length of the camera axis will always be the same and never change. This way the character will always have the same distance from the camera. When the camera axis has detected a collision, it will change the angle. For example if the camera is positioned above the character and the character is walking under a bridge, the camera axis will hit the bridge. This will make the camera axis change the angle. Since the collision detection occurred on top of the camera axis, the camera axis will be rotated downward and the camera will go underneath the bridge together with the character. When the camera has passed the bridge, the camera axis will go back to its original position above the character.

A problem that will occur with this camera management is at narrow passages where the camera axis is too long to fit within the world. For example if the character walks backwards into a wall, the camera will hit the wall before the character. With this camera management, the camera will rotate upwards along the wall. But if the ceiling is lower than the length of the camera, the camera will be stopped even if the character hasn't hit the wall. In this case the camera have to prevent the character's movement in order to keep the distance between the character and the camera. Another example is when the character has walked through a doorway into another sector while the camera is still in the old sector. If the character change direction and walks along the wall there will be a situation where the wall will be in the way between the character and the camera, thus not showing the character.

3.1.3 Flexible Distance

In this camera management the character, camera and the camera axis is affected. But unlike the camera management with fixed camera distance, the collision detection is checked on the character and camera while in same sector. The collision detection is checked on the character and camera axis if the camera and character is in different

sectors. Another difference is that the camera distance is flexible and can change. With the flexible camera distance the problem with the narrow passages from the previous camera management won't occur here. Also, the camera axis is only used when the character and the camera are in different sectors. The camera axis will help the camera to go through the portal that the character went to. But when the character and the camera is in same sector, the camera axis won't do anything. When the character and camera is in the same sector the length of the camera axis will be changed when a collision on the camera have been detected. When a collision have been detected and the character and the camera is in different sectors, the angle of the camera axis will be changed in the same way as the camera management with fixed camera distance.

A problem that occurs is when the character for example goes inside a pipe that lies on the floor in the same sector. The camera will continue its path outside the pipe and not displaying the character. A solution to this problem is to create the path inside the pipe as a new sector so that the camera axis will be affected and moved into the pipe together with the character. Another problem that occurs is when the character has its back near a wall or into a wall. The camera will only show the character's back or be inside the character and not displaying anything in front of the character. A solution to this problem could be to make the character transparent or semi-transparent when the camera is too close to the character. But then the character won't be visible. The camera points to the character's back. If the camera is above the character and the character have its back into a wall, the camera will point downwards from top of the character's head. Then the camera will only show what's underneath the character and not in front of the character if the character is transparent in this situation. If the character isn't transparent in this situation, the camera will only show the head of the character.

3.1.4 Combining Fixed and Flexible Distance

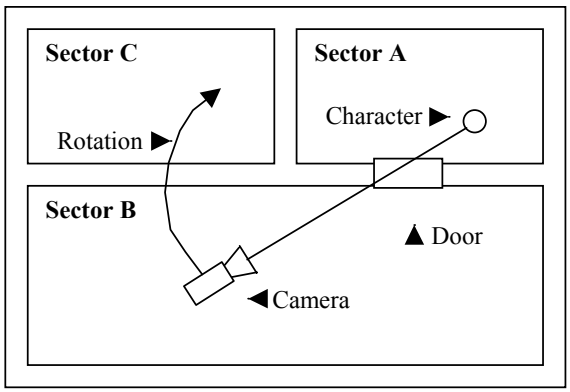
The last camera management will be a combination of a fixed and flexible camera distance. The camera will be managed just like camera management with fixed camera distance except when the camera is stuck. When the camera gets stuck for example in a narrow passage, the camera distance will become flexible and change so that the character will still be able to move. While the camera isn't stuck, the camera distance will be fixed.

The problem with camera management with fixed camera distance will be solved with this management. But the same problems will occur as camera management with flexible camera distance in the situation where the camera distance has to change. But this can be solved by making the character invisible when the camera is near the character like stated before.

3.2 Crystal Space Sectors

The character and camera must be moved through a portal in order to change sector. The character and camera will remain in the current sector until passing through a portal. This creates a problem when the view is a third-person view. Because there is a distance between the character and camera, there can occur a situation where the character and the camera are located in different sectors. If the character for example has passed a doorway with a portal to another sector the character and the camera will have different sectors. If the character then rotates left or right the camera will also rotate, but it will go through the wall instead of the portal because the distance between the character and the camera is longer than the width of the doorway. The camera will not go through the portal and will remain in the old sector, in spite of that it is outside the old sector. The camera will only display objects within its sector and that is visible through a portal, so the camera will not display objects outside its sector. A possible solution to this problem is to change the sector of the camera to the sector of the character. But then the problem of when to change the sector occurs. It is not possible in Crystal Space to determine if the camera is out of its sector. One possible way to

change the sector could be that the sector of the camera will be changed when it's different from the sector of the character and a collision have been detected. But if a collision has been detected and the camera is still in the sector, the sector of the camera will be changed to the sector of the character, which is wrong sector. There is another problem that will occur if the sector of the camera will be changed to the sector of the character when the camera is outside its sector. The problem occurs when there are three sectors involved. For example the camera is in sector B and the character is in sector A. When character is rotating the camera will be outside sector B and arrives to sector C, the third sector. If the camera changes the sector at that point to sector A, which is the sector of the character, the sector of the camera will be wrong since its real sector is sector C.



A problem that occurs with Crystal Space sectors is the use of collision detection. The collision detection is only checked within a sector. An object can't have two sectors, so when the character and the camera is in different sectors, the camera axis will be in two sectors. Checking the collision detection on the camera axis will only check in one sector, so the camera axis will go through the objects and walls in the other sector. This occurs for example when going through a doorway with a portal between two sectors. The collision detection on the front part of the camera axis, which is in the first sector, will functional correctly. But the back part of the camera axis will be in the second sector and not detect any collision, and therefore go through the wall above the doorway.

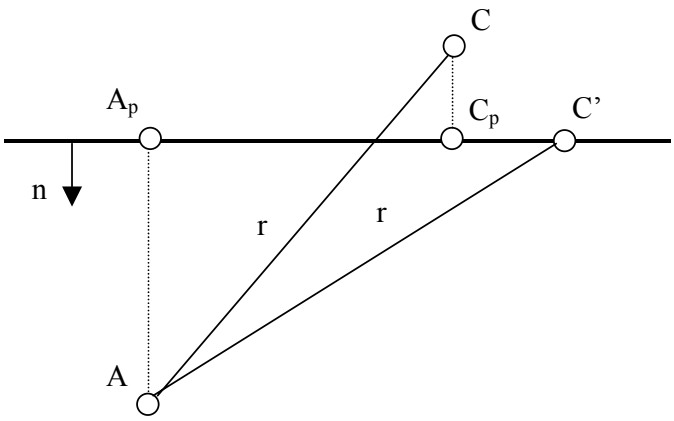
3.3 Mathematic Calculations

The mathematic calculations of the camera management used in the demo application are described here. The camera management used here is to change the position of the camera when the camera collides with an object such as a wall so that the camera is moved out from the wall. This way, the camera is always inside its own sector.

Point A in the calculations stands for the position of the character and point C stands for the original position of the camera. The new position of the camera is point C'. The length r is the fixed length between the camera and the character.

3.3.1 Description

The thick horizontal line in the figure is the plane seen from the side. The normal is pointed downwards and have the character n. There is two points A and C in either side of the plane. The length between the points A and C is r and crosses the plane. The vector between A and C have the formula $|\overline{AC}|= r$. There is another point called C' that is located on the plane. The length between point A and C' is also r, which results in $|\overline{AC'}|= r$. The projection of point A on the plane has a point A_p. Also, the point C_p is a projection on the plane from point C.



3.3.2 Formulas

The values that are known are the length r, normal n and the points A and C. Also a point P in the plane is known. With these values the projection points A_p and C_p can be calculated. With help from the projection points the point C' can be found. A general calculation will be made so these values are used:

A = (A_x, A_y, A_z)
C = (C_x, C_y, C_z)
P = (P_x, P_y, P_z)
n = (n_x, n_y, n_z)

To calculate the projection points A_p and C_p, the projection formula is needed. The projection formula is:

$$u' = u - \frac{u \bullet n}{n \bullet n} * n$$

Using the projection formula, the projection point A_p is calculated:

$$A_p - P = (A - P) - \frac{(A - P) \bullet n}{n \bullet n} * n$$

$$A_p = (A_x, A_y, A_z) - P + P - \frac{((A_x - P_x) * n_x + (A_y - P_y) * n_y + (A_z - P_z) * n_z)}{1} * (n_x, n_y, n_z)$$

For simplicity, the variable w is used and inserted into the formula.

$$w = ((A_x - P_x) * n_x + (A_y - P_y) * n_y + (A_z - P_z) * n_z)$$

The projection point Cp is calculated in the same way giving these formulas:

$$A_p = (A_x - w * n_x, A_y - w * n_y, A_z - w * n_z)$$

where $w = ((A_x - P_x) * n_x + (A_y - P_y) * n_y + (A_z - P_z) * n_z)$

$$C_p = (C_x - v * n_x, C_y - v * n_y, C_z - v * n_z)$$

where $v = ((C_x - P_x) * n_x + (C_y - P_y) * n_y + (C_z - P_z) * n_z)$

To calculate the point C', a formula using the projection points and a delta value can be used. The formula looks like this:

$$C' = (1-\lambda) * A_p + \lambda * C_p$$
$$C' = A_p + \lambda * (C_p - A_p)$$

This formula will be inserted in the vector $\overline{AC'}$ so that the delta value can be calculated.

$$\overline{AC'} = C' - A$$
$$\overline{AC'} = (A_p - A) + \lambda * (C_p - A_p)$$

To make the formula easier to follow and understand, the variables s and t is used in the formula, replacing these calculations:

$s = A_p - A$
$t = C_p - A_p$

Inserting these variables into the formula will give this simplicity:

$$\overline{AC'} = s + \lambda * t$$

It is known that the length of the vector $\overline{AC'}$ is equal to the length r, which is a known value. Therefore the length r is used to create an equation that will help solving the delta value. The previous calculated formula will be inserted into this equation.

$$r = |\overline{AC'}|$$
$$r^2 = (s + \lambda * t) \bullet (s + \lambda * t)$$
$$\lambda^2 + 2 * \frac{s \bullet t}{t \bullet t} * \lambda + \frac{s \bullet s - r^2}{t \bullet t} = 0$$

The delta value can be calculated using the second-degree equation.

$$\lambda = -\frac{1}{2 * \frac{s \bullet t}{t \bullet t}} \pm \sqrt{\left(\frac{1}{2 * \frac{s \bullet t}{t \bullet t}} \right)^2 - \frac{s \bullet s - r^2}{t \bullet t}}$$

The equation will give two values for the delta value. The one that is closest to the point C' will be used. Using the delta value and the projection points A_p and C_p, the point C' can be calculated with this formula:

$$C' = A_p + \lambda * (C_p - A_p)$$

3.3.3 Calculation

To test the formulas, an example will be used. The known values are the normal and the two points:

A = (1, 0, 0)
C = (-2, 5, 0)

P = (-1, 0, 0)
n = (1, -1, 0)

The length between the points A and C is calculated with a formula.

$$r = |\overline{AC}|$$
$$r = |(-2,5,0) - (1,0,0)|$$
$$r = \sqrt{34}$$

Dividing with the square root of two normalizes the normal.

$$n = \left(\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}, 0\right)$$

The projection points are calculated.

$$A_p = (A_x - w * n_x, A_y - w * n_y, A_z - w * n_z),$$
where $w = (A_x * n_x + A_y * n_y + A_z * n_z)$
$$C_p = (C_x - v * n_x, C_y - v * n_y, C_z - v * n_z),$$
where $v = (C_x * n_x + C_y * n_y + C_z * n_z)$.

$$w = (1 - (-1)) * \frac{1}{\sqrt{2}} + (0 - 0) * \frac{-1}{\sqrt{2}} + (0 - 0) * 0$$
$$w = \frac{2}{\sqrt{2}}$$

$$v = (-2 - (-1)) * \frac{1}{\sqrt{2}} + (5 - 0) * \frac{-1}{\sqrt{2}} + (0 - 0) * 0$$
$$v = \frac{-6}{\sqrt{2}}$$

$$A_p = \left(1 - \frac{2}{\sqrt{2}} * \frac{1}{\sqrt{2}}, 0 - \frac{2}{\sqrt{2}} * \frac{-1}{\sqrt{2}}, 0 - \frac{2}{\sqrt{2}} * 0\right)$$
$$A_p = (0,1,0)$$

$$C_p = \left(-2 - \frac{-6}{\sqrt{2}} * \frac{1}{\sqrt{2}}, 5 - \frac{-6}{\sqrt{2}} * \frac{-1}{\sqrt{2}}, 0 - \frac{-6}{\sqrt{2}} * 0\right)$$
$$C_p = (1,2,0)$$

Now that all the necessary values is calculated, the delta value will now be determined with help from the second degree equation formula.

$$\lambda = -\frac{1}{2 * \frac{s \bullet t}{t \bullet t}} \pm \sqrt{\left(\frac{1}{2 * \frac{s \bullet t}{t \bullet t}}\right)^2 - \frac{s \bullet s - r^2}{t \bullet t}}$$

$$s = A_p - A = (0,1,0) - (1,2,0) = (-1,1,0)$$
$$t = C_p - A_p = (1,2,0) - (0,1,0) = (1,1,0)$$

$$s \bullet t = (-1,1,0) \bullet (1,1,0) = 0$$
$$s \bullet s = (-1,1,0) \bullet (-1,1,0) = 2$$
$$t \bullet t = (1,1,0) \bullet (1,1,0) = 2$$

$$\lambda = -\frac{1}{2 * \frac{0}{2}} \pm \sqrt{\left(\frac{1}{2 * \frac{0}{2}}\right)^2 - \frac{2 - \sqrt{34}^2}{2}}$$
$$\lambda = \pm 4$$

From the two values the delta value has, $\lambda = 4$ is used. The point C' is calculated with help of the delta value.

$$C' = A_p + \lambda * (C_p - A_p)$$
$$C' = (0,1,0) + 4 * ((1,2,0) - (0,1,0))$$
$$C' = (4,5,0)$$

To test if the C' value is correct, the value is inserted in these formulas:

$$(C' - P) \bullet n = 0$$
$$((4,5,0) - (-1,0,0)) \bullet (1,-1,0) = 0$$
$$0 = 0$$

$$|\overline{AC'}| = |\overline{AC}|$$
$$|(4,5,0) - (1,0,0)| = |(-2,5,0) - (1,0,0)|$$
$$\sqrt{34} = \sqrt{34}$$

This proves that C' is correct.

3.4 Summary

The best technique to manage an over-the-shoulder view is to use a combination of fixed and flexible camera distance since that would solve most problems. The other camera management techniques have more disadvantages. The main implementation is to create a fixed camera distance and make the camera move to the best position when a collision has been detected on the camera, and in use a flexible camera distance in special cases such as narrow paths.

The formula for finding the new position of the camera after a collision has been calculated will be used in the application. The formula will move the camera to a position in its own sector and still have a fixed distance between the camera and the character.

Chapter 4

Implementation

The implementation will be performed in three steps so that the development will become easier by only concentrating on a few tasks. This way it will be easier to understand the development and testing. These three steps that will be performed are the following:

Create a Crystal Space application that will display the Cal3D Character.
Create a Crystal Space plug-in that will display the Cal3D Character.
Create a Crystal Space application that will test the plug-in.

On the first step the focus will be on the Crystal Space objects and the Cal3D library. A simple Crystal Space application will be created and the Crystal Space map files will be used. On the second step the focus will be on creating a working plug-in for Crystal Space and using the Cal3D loader from previous step. The last step will test the plug-in created from second step. This step will be a demonstration of a character that can be controlled by the user from the keyboard. The user can move around the character in a Crystal Space world.

The Cal3D project includes besides the library also some data files and a couple of applications created with OpenGL that demonstrates the Cal3D library. The data files are three Cal3D characters called Cally, Paladin and Skeleton. These characters will be used to test the applications that will be implemented. The Cally and Skeleton characters don't have any textures, but only some color materials. The Paladin character has several different textures. So the Paladin character will be the main test object.

The implementation will be on Crystal Space version 0.94 that was released 8 April 2002 and Cal3D version 0.7 that was released 5 January 2002.

4.1 Cal3D Viewer

The first implementation step will be to display a character created with the Cal3D library on a Crystal Space environment. An application will be constructed so it will read a Cal3D character from an external file and display it on the screen. This application will be called the Cal3D Viewer.

The Cal3D Viewer can be divided into two tasks. The first task will be to create a Crystal Space application that will display a mesh object. The second task will be to import the Cal3D character and display it as a mesh object so that the Crystal Space engine can take care of it.

4.1.1 Crystal Space Application

When creating a Crystal Space application, there are a few minimum requirements that must be included. Every source file must import the header file *cssysdef.h*. This header file must be included first before the other header files. After including all header files, the macro CS_IMPLEMENT_APPLICATION shall be used in one source file. It controls if the main function is correctly linked and called on every platform.

The first thing that will be created is a simple Crystal Space application. Besides the constructor and destructor, the application needs an initialization function and a start function. The initialization function will setup Crystal Space and the application. The start function will run the application. All these four functions will be public functions. There will also be a function that will initialize a frame and a function that will draw a frame. These will be private functions and called SetupFrame respective FinishFrame. All these functions will be included when creating the Cal3D Viewer application. The main function in the application looks like this:

```
int main ( int argc, char* argv[] )
{
        cal3dviewer = new Cal3DViewer ();

        if ( cal3dviewer->Initialize ( argc, argv ) )
                cal3dviewer->Start ();

        delete cal3dviewer;

        return 0;
}
```

The main function will create an instance of the Cal3D Viewer class. Then it will initialize the application and Crystal Space by calling the initialization function on the instance.

```
bool Cal3DViewer::Initialize ( int argc, const char* const argv[] )
{
        object_reg = csInitializer::CreateEnvironment ( argc, argv );
        if ( !object_reg ) return false;

        if ( !csInitializer::RequestPlugins ( object_reg, ... )
                return false;
        ...

        if ( !csInitializer::OpenApplication ( object_reg ) )
                return false;

        ...

}
```

The first thing that has to be initialized is the environment. The environment includes the object registry that is returned, the SCF and the creation of some entities such as plug-in manager and event queue. After the initialization of the environment, all the requested plug-ins has to be loaded. The function RequestPlugins will first load the plug-ins that comes from the command line. After that it loads all the plug-ins from the configuration file and lastly it loads all the plug-ins that is requested in the parameters of the function RequestPlugins. After that the object registry will be queried to get a reference from all the objects that will be used in the application. After all references have been received, the window will be opened with the function: OpenApplication. If the initialization didn't return any important errors, the main function will call the start function on the Cal3D Viewer class instance.

```
void Cal3DViewer::Start ()
{
        csDefaultRunLoop ( object_reg );
}
```

The start function will simply run the default main loop. The main loop will initialize the frame and then draw the frame all the time, until the application terminates. The event handler checks if the user wants to terminate the application by checking if the key that terminates the application is pressed. If the application will be terminated, the event handler uses the object registry to find the global event queue object. It will then send a quit message to all interested parties so that they can clean up the memory and then end the main loop.

4.1.2 Crystal Space Object

Two functions will be added on the application to display a mesh object in a Crystal Space world. The first function will load a Crystal Space world from an external file and will be called LoadMap. The second function will be called LoadObject and will load a mesh object from an external file. The external file will be a Crystal Space Mesh Factory.

```
bool Cal3DViewer::LoadMap ( char* map_name, char* dir_name )
{
        iVFS* VFS = CS_QUERY_REGISTRY ( object_reg, iVFS );
        VFS->ChDir ( dir_name );
        VFS->DecRef ();

        If ( !loader->LoadMapFile ( map_name ) )
                return false;

        engine->Prepare ();

        if ( engine->GetCameraPositions ()->GetCount () > 0 )
        {
                ...
        }

        ...
}
```

The map will be loaded from the virtual file system (VFS). So the first thing that has to be done is to receive the VFS library and change the current directory to the directory where the map file is located. After that, the map file is loaded into the memory by the function LoadMapFile. By calling Prepare on the engine, the engine will update all necessary values and making sure that all lights and textures are correctly loaded and registered into the game engine. When the map is part of the engine, the camera has to be moved to the right position. The camera will be set to the first camera location in the map file that can be found, else it will get a standard starting position.

```
bool Cal3DViewer::LoadObject ( char* object_name, char* object_location, ... )
{
        ...

        // Load a texture
        iTextureWrapper* txt = loader->LoadTexture ( ... );
        ...

        // Load a sprite
        iMeshFactoryWrapper* imeshfact = loader->LoadMeshObjectFactory ( ... );
        ...

        // Create the sprite
        iMeshWrapper* sprite = engine->CreateMeshWrapper(..., object_name, ...);

        ...

}
```

The first thing that is needed when loading an object is to load a texture that will be used for the mesh object. The texture will be loaded from an external file. The name and location of the texture is specified in the parameters of the LoadObject function and must be the same name as the material name specified in the mesh factory file that will be loaded. After that, the mesh object will be loaded from the file specified in the parameters. When calling CreateMeshWrapper, the loaded mesh object will be created and registered in the engine. After that, the characteristics of the mesh object can be set such as lighting.

```
// Get avatar
iMeshWrapper* mesh =  engine->FindMeshObject( object_name );
if( !mesh )
return false;

mesh->GetMovable()->SetTransform( current_transform );
mesh->GetMovable()->SetPosition( current_sector, current_position );
mesh->GetMovable()->UpdateMove();
```

An object that has been loaded can be requested from the engine by using the FinMeshObject function and specify the name of the object. After the object have been found, various characteristics of that specific object can be changed for example the rotation and position of the object. Using the function UpdateMove will update all changes into the game engine.

4.1.3 Cal3D Library

The management of the Cal3D character will be on three main functions. There will be a function that loads the character, a function that initializes the character and a function that converts the character into a mesh object. The first thing is to load the character from the external file. Since the character is stored in Cal3D format and not in mesh factory format, the LoadObject function will not be used. The Cal3D file will be loaded and then converted into a mesh object so that the Crystal Space engine can handle the object.

```
bool Cal3DViewer::LoadModel ( const std::string& str_filename )
{
        // Open file
        std::ifstream file;
        file.open ( str_filename.c_str (), ... );

        // Create core model instance
        if ( !core_model.create ( "avatar" ) ) return false;

        // Parse all lines from the config file
        for ( int line = 1; ; line++ )
        {
                // Read next line
                std::string str_buffer;
                std::getline ( file, str_buffer );

                // Error handling
                ...

                // Get the key
                std::string str_key = str_buffer.substr ( ... );

                ...

                // Get the data
                std::string str_data = str_buffer.substr ( ... );

                // Create model
                if ( str_key == "skeleton" )
                {
                        if ( !core_model.loadCoreSkeleton ( str_data ) )
                                return false;
                }

                ...
```

```
        }

        ...
}
```

The first thing that the LoadModel function does is open the configuration file of the Cal3D character. The core model, which is a global variable, will be used for storing the character. An empty core model is first created before the data from the external file is stored. After that, the opened file will be read line by line and its content will be checked. Some error handling will be made such as checking the end of file, checking if the read line is valid and skipping white spaces and comment lines. When all error handling is done, the key value and data value will be stored. For example if the read line is:

skeleton=paladin.csf

Then the key value will be 'skeleton' and the data value will be 'paladin.csf'. When both values are stored, the key value will be checked. In this case, the key value is 'skeleton' which means that the data value points on a skeleton data file. The function loadCoreSkeleton will be called on the core model with the data value as parameter, which will load the skeleton data file in the core model. This will be repeated until every line has been read on the file. When the LoadModel is finished, the character has been stored correctly into the core model.

```
bool Cal3DViewer::InitModel ()
{
        // Create material thread
        for ( int material_id = 0; ... )
        {
                ...
        }

        // Create model instance
        if ( !model.create ( &core_model ) ) return false;

        // Load textures
        for ( int texture_id = 0; ... )
        {
                ...

                for ( int map_id = 0; ... )
                {
                        // Get filename
                        std::string name = core_material->getMapFilename (...);

                        // Load texture
                        iImage* image = LoadTexture ( name );

                        // Store texture
                        core_material->setMapUserData ( map_id, image );
                }
        }

        // Attach all meshes
        for ( int mesh_id = 0; ... )
                model.attachMesh ( mesh_id );

        model.setMaterialSet ( 0 );

        ...

        model.update ( 0.0f );

        return true;
}
```

The Cal3D library itself does not handle textures because there are different ways to manage the textures in applications. The texture management must be done in the application. But the Cal3D library supports a flexible system for texture handling. For

handling the textures in Cal3D the materials are grouped in two different ways. For each core model there is one or more material sets and additional one or more material threads. The material sets are grouped by a common look such as ‘skin’ or ‘leather’, while the material threads are grouped by a specific part of the model such as ‘torso’ or ‘helmet’. Either one could be used. The first thing the InitModel function does is to initialize all the material threads. After that a model is created from the core model. Because the Cal3D library doesn’t manage the textures, this is done in the InitModel function. Every material has one or more maps. Each map has a filename of a texture associated to it. What the InitModel function does is to go through all maps to receive the filename for the texture. It then loads the texture from the file and stores it by calling setMapUserData on the core material. At start there are no meshes on a newly created model, so all the meshes have to be attached to the model by calling attachMesh on the model. There is no material set attached to the model either so calling setMaterialSet on the model will attach a material set to the model. When all values have been set, it’s necessary to call update on the model to calculate the new pose of the skeleton.

```
bool Cal3DViewer::CreateModel ()
{
        CalRenderer* renderer = model.getRenderer ();
        if ( !renderer->beginRendering () ) return false;

        // Go through all meshes of the model
        for(int mesh_id = 0; mesh_id < mesh_count; mesh_id++)
        {
                // Go through all submeshes of the mesh
                for(int submesh_id = 0; submesh_id < submesh_count; submesh_id++)
        {
                // Select the mesh and submesh for further access
                if(renderer->selectMeshSubMesh(mesh_id, submesh_id))
                {

                        // Receive data from the renderer
                        int vertex_count = renderer->getVertices( ... );
                        int normal_count = renderer->getNormals( ... );
                        int texture_count = renderer->getTextureCoords( ... );
                        int face_count = renderer->getFaces( ... );

                        ...

                        // Create mesh factory
                        iMeshFactoryWrapper* mesh_factory = ...

                        // Create 3D sprite factory
                        iSprite3DfactoryState* sprite = ...

                        // Set data into 3d sprite state

                        sprite->SetMaterialWrapper( ... );
                        sprite->AddVertices( vertex_count );

                        // Add all vertices
                        for(int vertex_index = 0; ... )
                                sprite->SetVertex( ... );

                        // Add all normals
                        for(int normal_index = 0; ... )
                        {
                                sprite->SetNormal( ... );

                                ...

                                // Register object into the engine
                                iMeshWrapper* mesh_object = ...

                                // Resize and rotate object
                                ...
                        }
                }
        }

        renderer->endRendering ();
```

```
        return true;
}
```

The Cal3D character will be created as a Crystal Space mesh object and registered into the engine so that the engine can draw the character. The Cal3D library doesn't do the rendering of the character, but does support an easy way to do it with something called the Renderer. The CreateModel function uses the Renderer to create the character. The first thing that is done is receiving the Renderer and start it. The character has one or more meshes, and each mesh have one or more submeshes. So each submesh have to be selected. From each submesh, the vertices, the normals, the texture coordinates and the triangles is received and stored in local variables. Now a Crystal Space Mesh Factory has to be created. One texture is received from an external file and stored to the mesh factory. A Sprite3D state from the mesh factory is received. All vertices, normals, texture coordinates and triangles from the character will be set in the Sprite3D state. The created mesh factory will be registered to the engine so that the engine can draw the character. There is a problem with the character orientation: the coordinate system on Crystal Space has the Z-axis inwards the screen and the X-axis and Y-axis is the screen, while the Cal3D coordinate system has the Y-axis inwards the screen and the Z-axis and X-axis is the screen. To fix this, the character has to be rotated by rotating every vertex along the X-axis. The Cal3D character is also too big for Crystal Space, so the character size is reduced.

A disadvantage with the Crystal Space mesh object is that it can only use one texture for each object. So the Cal3D character will only have one texture. A solution to this problem would be to implement a new Crystal Space mesh object, which will be done when creating the Cal3D plug-in.

4.2 Cal3D Plug-In

Crystal Space is based on plug-ins. To make it easier to use the Cal3D implementation in a Crystal Space project, the implementation will be created in form of a Crystal Space plug-in. Creating a plug-in for the Cal3D implementation will also fix the texture problem from the Cal3D Viewer application. The problem in Cal3D Viewer was that only one texture could be used for each character.

The Cal3D plug-in will implement three interfaces. The interfaces are iCal3DState interface, iSprite3DState interface and iPolygonMesh interface. The iCal3DState interface will be its own interface of the Cal3D plug-in, while the other two interfaces come from Crystal Space.

Besides the interfaces, the Cal3D plug-in also includes all the necessary functions that the Crystal Space needs in order to run the plug-in within the game engine. Some of the functions that the Crystal Space needs are:

Draw:	Which is called for each frame to render and update the character.
DrawTest:	Which is used for clipping the objects within the viewing screen.
UpdateLightning:	Which is used to set the lightning on each submesh of the character.

The function Draw is designed the same way as the CreateModel function from the Cal3DViewer application. The loading and initialization of the model from Cal3DViewer is also included into the Cal3D plug-in and works the same way.

4.2.1 iCal3DState Interface

The iCal3DState will be the own interface of the Cal3D plug-in. This interface will be used to change the animation sequence of the character. It will also be used for loading the character into memory and setting the LOD-level. This interface takes care of the Cal3D library.

There are seven functions that the iCal3DState interface has. These are LoadMesh, SetAction, ExecuteAction, BlendAction, ClearAction, ClearAllActions and SetLodLevel. 'Action' indicates an animation sequence of the character.

```
bool LoadMesh ( const char* filename, iLoader* loader, iEngine* engine )
{
	if( !load_model( filename ) )
		return false;

	for( int materialId = 0; ... )
	{
		// Get the core material.
		core_material = core_model.getCoreMaterial( materialId );

		// Go through all the maps of the core material.
		for( int mapId = 0; ... )
		{
			// Receive the texture
			std::string strName = core_material->getMapFilename(...);
			iImage* image = load_texture( strName );

			...

			// Save texture in the core model
			core_material->SetMapUserData( mapId, ... );
		}
	}

	create_boundingbox();

	return true;
}
```

The LoadMesh function will load the Cal3D character from a file. It will open the configuration file of the character and load the skeleton, the meshes, the animations and the materials. If no textures are attached to the character, a standard texture will be used. Also the bounding box used for collision detection is created for the character.

```
bool SetAction ( const char* name, float delay, float delay_new, float weight)
{
	for ( int index = 0; index < _anim_count; index++ )
		if ( _actions[index] == std::string ( name ) )
		{
			model.getMixer ()->clearCycle( _current_anim, delay );
			model.getMixer ()->blendCycle( index, weight, delay_new );
			_current_anim = index;
			return true;
		}

	return false;
}
```

The name of the action from the parameters will be searched in the array, which holds all the names of the actions the character can perform. If no match was found, the function will return false. The most recently animation cycle that is running on the character will be removed if a match is found and the character will run the new animation cycle. Current animation cycle will be set to the new animation cycle. The current animation will loop until stopped with the ClearAction or ClearAllActions functions.

When changing an animation, the weight of the animation can be set. Also the fading can be set by the delay parameters.

```
bool ExecuteAction ( const char* name, float delay_in, float delay_out )
{
	for ( int index = 0; index < _anim_count; index++ )
		if ( _actions[index] == std::string ( name ) )
		{
			model.getMixer ()->executeAction ( index, delay_in, ... );
			_current_anim = index;
			return true;
```

```
                }

        return false;
}
```

Just like the SetAction function, the name of the action from the parameters will be searched in the ExecuteAction function. This function will run the new animation and set the animation to the current one. The animation will only be running once without looping. The animation can't be interrupted with the functions ClearAction or ClearAllActions.

```
bool BlendAction ( const char* name, float delay, float weight )
{
        for ( int index = 0; index < _anim_count; index++ )
                if ( _actions[index] == std::string ( name ) )
                {
                        model.getMixer ()->blendCycle ( index, weight, delay );
                        _current_anim = index;
                        return true;
                }

        return false;
}
```

The BlendAction function will interpolate the weight of an animation cycle. The current animation will be set to the new animation. The difference between the BlendAction function and SetAction function is that BlendAction doesn't delete the most recently animation cycle.

```
bool ClearAction ( const char* name, float delay )
{
        for ( int index = 0; index < _anim_count; index++ )
                if ( _actions[index] == std::string ( name );
                {
                        model.getMixer ()->clearCycle ( index, delay );
                        return true;
                }

        return false;
}
```

The ClearAction function will remove a selected animation cycle that is running. Fade out an animation cycle in a given amount of time. It has no effect on animations that are not running and animations that have been set with the function ExecuteAction.

```
void ClearAllActions ()
{
        for ( int index = 0; index < _anim_count; index++ )
                model.getMixer ()->clearCycle ( index, 0.0f );
}
```

The ClearAllActions function will remove all animations that are currently running with an exception from animation sequences set with the function ExecuteAction. All animation cycles that are removed will be removed without fading.

```
void SetLodLevel (  float lod_level )
{
        model.setLodLevel ( lod_level );
}
```

The SetLodLevel function will change the LOD-level of the character, which can increase the performance. The LOD-level will be changed on all attached meshes.

4.2.2 iSprite3DState Interface

To describe the API for the sprite 3D mesh object, Crystal Space uses iSprite3DState. The interface of the Sprite3DState will be implemented in the Cal3D plug-in. This interface will be included in the Cal3D plug-in because it's a common interface for

mesh objects. Including the iSprite3DState interface will keep the high integration between plug-ins that Crystal Space have.

The iSprite3DState interface includes several functions, but only one will be implemented for the Cal3D plug-in. The function that will be implemented is the SetAction function. The rest of the functions will still be included into the source code since it's an interface, but they won't do anything. But since they are already included, they can be implemented if needed.

The SetAction function from iSprite3DState will actually call the iCal3DState function with the same name. The SetAction prototype in iSprite3DState looks like this:

bool SetAction (const char* name);

The SetAction function in iSprite3DState uses only one parameter while the SetAction function in iCal3DState uses four parameters. When using the SetAction function from iSprite3DState, the rest three parameters will have the standard value 1.0.

4.2.3 iPolygonMesh Interface

Crystal Space has a collision detection plug-in called iCollider. The iCollider uses the iPolygonMesh interface from each object to check a collision. A mesh object that doesn't have the iPolygonMesh interface implemented are considered as an object with no collision detection by the engine. The iPolygonMesh interface will be implemented in the Cal3D plug-in so that the Cal3D character can be tested for collision with the surroundings.

To test the collision of the character, a bounding box will be used. The reason why to use a bounding box instead of the character itself, is because of the performance. Using collision detection on the character itself involves testing every triangle on the character, which means a lot calculation will be done. To improve the performance, a bounding box will be used instead.

The bounding box will be created once at initialization of the character. The bounding box will be calculated from all the vertices of the character. Crystal Space has a library called csBox3, which calculates a box from a number of vertices. This class will be used when creating the bounding box. The vertices and polygons of the bounding box will be received from that class.

The iPolygonMesh interface includes four functions and all four will be implemented in the Cal3D plug-in. They are GetVertexCount, GetVertices, GetPolygonCount and GetPolygons.

```
int GetVertexCount ()
{
        return 8;
}
```

This function will return the number of vertices for the character. Since the collision detection will be on a bounding box, the vertices will be each corner of the box. There are eight corners on a box so there will be eight vertices.

```
CsVector3* GetVertices ()
{
        return _bounding_box_vertices;
}
```

This will return the pointer to the global array of vertices. When the initialization of the bounding box was made, each vertex was stored on a global variable, which are returned in this function.

```
int GetPolygonCount ()
{
        return 6;
}
```

This function will return the number of polygons for the character. The bounding box has six polygons, one for each side of the box.

```
CsMeshedPolygon* GetPolygons ()
{
        return _bounding_box_polygons;
}
```

This will return the pointer to the global array of polygons. The polygons of the box were stored globally during the initialization of the bounding box.

4.3 Demo Application

All features of the implemented Cal3D plug-in must be tested to see if they are working correctly. A Crystal Space application that uses the Cal3D plug-in shall be created to test the plug-in. The main functions of the application will be implemented just like the Cal3D Viewer such as main function, initialization function, event handler, the start function, the frame initialization function and frame drawing function. These functions are the base of a Crystal Space application.

The application will also be a demonstration of a character that can be controlled by user input. The user shall have the possibility to move around the character in a Crystal Space world and change the actions (animations) of the character such as walking, running and waving. The movement of the character can be divided into two sections. The first section is movement of the character and the second section is movement of the camera.

With help of the LoadMap function from the Cal3D Viewer application, it's possible to load any Crystal Space world map to test the character in. It will also be possible to load any Cal3D character to be used in the application.

Both the character movement and the camera movement are implemented in the SetupFrame function that is called before each frame. The camera integrates with the character in order to keep the character within the viewing sight. For each frame, the position and rotation for the character and the camera is updated and changed depending on the user input or the collision detection.

4.3.1 Character Movement

The user input will be performed from a keyboard. The keys that have an effect on the character is the following:

Up arrow:	Moves the character forward.
Left arrow:	Turns the character to the left.
Down arrow:	Moves the character backwards.
Right arrow:	Turns the character to the right.
DEL key:	Moves the character to the left.
END key:	Moves the character to the right.
SHIFT key:	Makes the character running.
SPACE key:	Executes the 'waving' animation.
TAB key:	Executes the 'shoot arrow' animation.

The character will walk around the world with the arrow-keys and will strife with the DEL and END keys. Holding down the SHIFT key and moving the character around the world, will make the character running. Pressing for example the SPACE key while the character is running will combine the running animation and waving animation so that

the character is running and waving at the same time. Other combinations can be used to combine different animation sequences. Since the waving animation and the shoot arrow animation uses the same parts of the character body to animate, the animation looks strange when combining those two.

The input keys will be checked before each frame that is drawn, so the SetupFrame function will check if the user has pressed a key that affects the character. The values of the character will also be updated in the SetupFrame function.

```
void SetupFrame ()
{
        ...

        // Get avatar
        iMeshWrapper* avatar_mesh = engine->FindMeshObject ( OBJECT_NAME );

        // Avatar movement
        if ( kbd->GetKeyState ( CSKEY_RIGHT ) )
        {
                avatar_moves = true;
                avatar_rotation.RotateThis( CS_VEC_ROT_LEFT, rotation_speed );
        }
        ...

        // Get avatar state
        iCal3DState* avatar_state = SCF_QUERY_INTERFACE ( ... );
        if ( kbd->GetKeyState ( CSKEY_SPACE ) )
        {
                avatar_state->ExecuteAction( "paladin_wave" );
        }
        ...

        // Update avatar
        avatar_mesh->GetMovable()->SetTransform( avatar_rotation );
        avatar_mesh->GetMovable()->SetPosition( avatar_sector, avatar_position);
        avatar_mesh->GetMovable()->UpdateMove();

        // Check avatar collision
        if ( CheckCollision ( avatar_mesh, avatar_col ) )
        {
                ...
        }

        // Do gravity on avatar
        ...

        // Check gravity collision
        if ( CheckCollision ( avatar_mesh, avatar_col ) )
        {
                ...
        }

        ...
}
```

At initialization, the character is registered into the engine. The first thing that has to be done is to find the character from the engine and store it into a mesh wrapper. Then the user input is checked. If for example the user presses the up arrow on the keyboard, the values for moving the character forward are stored in local variables. The same thing is checked for the rest input keys including different states to update the animation of the character. When all input keys have been checked and all new character values have been stored in local variables, the values have to be updated. The old character values, which are used at collision detection, are saved in local variables before updating the new values. The new values are updated by setting the values into the mesh wrapper that was received from the engine. When the input values have been updated, the collision detection with the surroundings will be checked on the character with the new values. If a collision has occurred, the old values will be put back into the mesh wrapper. A very simple gravity is calculated on the character. The character will be moved down and new collision detection will be tested to see if the character collides with the floor or an object below the character.

4.3.2 Camera Movement

To see the character while moving around the world, a third-person camera is implemented. The camera has an over-the-shoulder view and follows the character movement. The character movement is completely independent on the camera position. This way the camera can't prevent the character from moving around when the camera collides with objects. The character can move around freely and have its own collision detection. But on the other hand, the camera is dependent on the character position. This way the character will always be visible through the camera view despite sectors, corners and other things that can stand in the way. Since the character should have the same size all the time on the screen, the distance between the character and the camera is always the same.

There are three values that specify the camera location in the world. These values are the camera length, the horizontal camera angle and the vertical camera angle. The camera length has the same value throughout the runtime of the application, but can be changed to obtain certain effects. The horizontal and vertical camera angles are used to change the orientation of the camera in the world. The horizontal angle lies within the X-plane and Z-plane and the vertical angle lies within the Y-plane and Z-plane.

```
void SetupFrame ()
{
        ...

        // Calculate camera position
        float camera_x = camera_dist * sin( angle_hor ) * cos ( angle_ver );
        float camera_y = camera_dist * sin( angle_ver );
        float camera_z = -camera_dist * cos( angle_hor ) * cos( angle_ver );
        camera_position = cam_axis_position + ( ... );

        csVector3 vA = cam_axis_position;
        csVector3 vC = camera_position;
        csVector3 vP;

        // Check collision
        iPolygon3D* collided_polygon = avatar_sector->HitBeam( vA, vC, vP );
        if( collided_polygon )
        {
                // Calculate the collision plane
                csPlane3 collision_plane = collided_polygon->GetWorldPlane();
                collided_plane.Normalize();

                // Check if A and C is on each side of the plane
                if( (vA-vP)*nomal * (vC-vP)*normal) < 0 )
                {
                        csVector3 vAp = Projection( collision_plane, vA );
                        csVector3 vCp = Projection( collision_plane, vC );

                        vA_dist = collision_plane.Distance( vA );
                        csVector3 vApCp = vCp - vAp;
                        vApCp.Normalize();
                        float t = sqrt(camera_dist*camera_dist - vA_dist*vA_dist);

                        // Update C with the new value
                        vC = vAp + t * vApCp;

                        // Move out camera from wall
                        vC = vC + -0.1 * normal;
                }
        }

        // Update camera location
        camera_transform.LookAt( vA - vC, csVector3(0,1,0) );
        camera_transform.SetOrigin( vC );
        view->GetCamera()->SetTransform( camera_transform );

        ...
}
```

The exact position of the camera is calculated with help of the horizontal and vertical camera angles and the distance between the character and the camera. The character point is set to the position where the camera points to and the camera point is set to the camera position.

When all is set, collision detection will be made with the HitBeam function. The camera itself doesn't have collision detection. But collision detection is checked between the camera and the character. This will help the camera to be located within its sector so it won't go through walls and other objects. HitBeam checks if a collision occurred between two points. If a collision is detected between the character point and the camera point, the collided plane is received and normalized. Before calculating a new camera position, it makes sure that the points are in each side of the collided plane.

If a collision has occurred between the character and camera points, the new value of the camera position is calculated. The camera point is updated with the new value, but since the new value is located inside the plane, it has to be moved out so that the camera can't see behind the plane. With help of the plane's normal, the camera position is moved out 0.1 units away from the plane. Lastly, the rotation and position of the camera is updated.

4.4 Summary

The Cal3DViewer was created as a basic Crystal Space application. The Cal3DViewer used Crystal Space's standard mesh object format for displaying a Cal3D character on a Crystal Space environment. But because of the limits of the mesh object, each object can only have one material attached to it, while a Cal3D can have as many as desired. The Cal3D characters didn't look as they suppose to look like on screen. This problem was fixed when the Cal3D plug-in was implemented.

Crystal Space uses a certain template for all plug-ins in order to keep the integration and compability between other components and plug-ins within the engine. This template was used when creating the Cal3D plug-in for Crystal Space. The Cal3D plug-in implemented the iSprite3DState interface because this interface is commonly used in Crystal Space projects. But only one function was implemented, a function to set an animation sequence. The main functions for the Cal3D plug-in were implemented into a completely new interface called iCal3DState. With help of this interface, it is possible to set and remove an animation sequence in different ways. Also the iPolygonMesh interface was implemented into the Cal3D plug-in to be used for collision detection of the character. There is a problem with the Cal3D plug-in and that is using Cal3D characters that don't use any textures. Instead of textures, the characters use material colors. Because of the limits in Crystal Space, it can't display special color lightning easily and correctly such as ambient colors, diffuse colors, specular colors and shininess. A texture has to be assigned to a mesh object in order for it to be correctly used. This problem might be fixed in future Crystal Space releases.

The features of the Cal3D plug-in are shown on a demo application that was implemented for two tasks: to test and to demonstrate the plug-in. The demo application uses a third-person camera that follows the character correctly. The distance between the camera and the character is fixed and collision detection is made to keep the camera within its sector. The collision detection for both the character and the camera will still be working even if the character and the camera are in different sectors.

The camera management in the demo application has some cases where it won't work correctly. These cases haven't been implemented into the application because there wasn't enough time to do it. One of these cases is when the camera is located between two planes. This can occur when the camera position first have to be moved out from the first plane, but doing that will position it behind another plane. The solution to this problem is to move out the camera position from both planes so that two possible positions are calculated. The position closest to the old camera position will be the new

position for the camera. Another case would be a collision between three planes for example two walls and the ceiling. In that case, the camera position will be located in the intersection point for all three planes. In this case the distance between the character and the camera can't be fixed because that would place the camera outside its sector. The last case, and the most critical, is when the area between the camera and the character is collided with a concave corner for example on the border of a doorway. The position of the camera will in this case be located on the plane that is defined by the collided corner and the position of the character.

References & Cast

Literature

Thalmann M. Nadia & Thalmann Daniel (1996) "Interactive Computer Animation"

Internet

Heidelberger Bruno 'Beosil', Cal3D, http://cal3d.sourceforge.net, 2002-08-15

Henry-Biskup Stefan, Anatomically Correct Character Modeling, http://www.gamasutra.com/features/visual_arts/19981113/charmod_01.htm, 2002-06-12

Keith Matt, The interverse Project, http://www.interverse.org, 2002-05-14

Ringuet JM, The Hardships of Animating Three-Dimensional Characters in Real Time Games, http://www.gamasutra.com/features/20010727/ringuet_01.htm, 2002-06-12

Tyberghein Jorrit, Crystal Space, http://crystal.sourceforge.net, 2002-04-20

Web3D Consortium Incorporated, H-Anim, http://h-anim.org, 2002-05-14

Worldforge, http://www.worldforge.org, 2002-08-15

Cast

Author	Arton Grajqevci
Project Supervisor	Lennart Ohlsson
Help with the Project	Calle Lejdfors
Help with the Project	Mathias Haage