Film Editor for True 3D Films

Ewan Borland
borlaned@dcs.gla.ac.uk
9803713

16th April 2002
LIST OF FIGURES

9 Interaction Package
9.1 OrbitBehavior .................................................. 23
9.2 OrbitModifier ................................................... 25
9.3 Actions .......................................................... 25

10 Capture Package
10.1 CaptureDialog .................................................. 26
10.2 ImageSourceStream and ImageDataSource .................... 27
10.3 FilmGenerator .................................................. 28
10.4 CaptureCanvas3D ............................................... 28
10.5 Capture Actions ............................................... 30

11 Testing .......................................................... 30

12 Evaluation ....................................................... 31

13 Acknowledgements ............................................... 31

List of Figures

1 Package hierarchy of the FilmEditor3D. .......................... 5
2 Package dependencies in the FilmEditor3D. ....................... 10
3 Internal class structure of the fe3d.background package. .......... 17
4 Internal class structure of the fe3d.sequence package. ............ 18
5 Internal class structure of the fe3d.interaction package. .......... 24
6 Internal class structure of the fe3d.capture package. ............ 26
7 Thread status of FilmGenerator ........................................ 29
1 Introduction

The 3D-MATIC Research Laboratory\(^1\) at the University of Glasgow\(^2\) has developed a system for dynamically capturing photo-realistic 3D models of a scene at frame rates suitable for the generation of 3D films. These *True 3D Films* are unique because they are not limited to a specific viewpoint. Each film is comprised of a sequence of 3D models and so can be viewed from any possible angle or range. Perhaps most significant is the ability of an observer to choose where he or she views the film from whilst playback is underway.

1.1 Problem Description

Viewing and interacting with these films is not a simple task. The films are generated as VRML\(^3\) files. There are a number of free and commercially available viewers for VRML files and most provide support for animation, but none that are currently available have animation as a central feature. As a result of this, current tools have proved slow or impossible to use for all but the most trivial of these 3D films, and user interaction tends to be severely limited whilst animation is underway.

In addition to this it is desirable to be able to perform some degree of sequencing on the films generated so that they may be combined, looped and otherwise edited in a similar fashion to their 2D counterparts. These films are also very important for demonstrating the work being done within the research group and as such it is desirable to be able to present them in a simpler format. Of particular interest is the ability to generate traditional 2D films from specific viewpoints, as these 2D films would not require any special visualization software.

To achieve these goals a new tool was needed which would be dedicated to the task of properly visualizing these *True 3D Films*. This project was intended to develop such a tool which would support the following initial requirements:

- Platform independence.
- Visualization of *True 3D Films* at realistic frame rates.
- Interactive camera behaviour.
- Capture of 2D films or still images for post-processing.
- Camera interaction whilst capturing to enable multiple 2D visualizations of one 3D film.
- Simple sequencing operations to allow films to be built up from arbitrary frames.
- Loading and saving of these 3D films, including any sequencing and camera position information which may have been added.
- Addition of simple background environments around the captured scene.

---

\(^1\)http://www.faraday.gla.ac.uk/.

\(^2\)http://www.gla.ac.uk.

\(^3\)Virtual Reality Modelling Language.
These requirements were repeatedly expanded and refined as the development process continued. However this refinement was done very informally at project meetings, and as such, many of the resulting requirements are simply clarifications of technical or user interface issues relating to the main requirements detailed above. Where they are significant these additional requirements will be documented at the appropriate point in the description of the implementation.

1.2 Java3D

It was suggested that this project should be implemented in Java3D\textsuperscript{4}. After detailed inspection of the features of the Java3D API this decision can be justified by noting the following advantages offered over other platforms:

- Efficient rendering engine included in API. So no need to worry about rendering.
- High level 3D scene manipulation which makes it easy to interact with 3D models or worlds.
- Close correspondence between graph structures and node types in Java3D and VRML simplify the loading/conversion process.
- Works on top of Java2 platform allowing easy GUI construction and event driven programming.
- Large number of utilities to simplify 3D application construction.
- Available for a wide variety of platforms.

Since platform independence was a stated requirement for the project this last point is clearly significant. Currently Java3D is available for the following platforms:

- SPARC Solaris
- Windows 98
- Windows Me
- Windows 2000
- Linux\textsuperscript{5}
- HP-UX\textsuperscript{6}
- AIX\textsuperscript{7}
- IRIX\textsuperscript{8}

\textsuperscript{4}For full details of the Java3D API consult http://www.java.sun.com/products/java-media/3D/.
\textsuperscript{5}Delivered by Blackdown http://www.blackdown.org/java-linux/jdk1.2-status/java-3d-status.html.
\textsuperscript{7}Delivered by IBM http://alphaworks.ibm.com/tech/aixj3d.
\textsuperscript{8}Delivered by SGI http://www.sgi.com/developers/devtools/languages/java3d121.html.
These advantages, coupled with a large degree of experience with the Java platform made Java3D the ideal choice. However, there were some additional considerations when choosing to use Java3D.

The API is a relatively new technology and at the time the project was started resources were limited to online tutorials provided by Sun and the Java3D API specification. That in combination with the lack of any identifiable Java3D expert within the Computing Department meant that the early stages of development involved a great deal of experimental work, to establish what could reasonably be accomplished within the constraints of the API.

## 2 Project Description

The tool produced as a result of this project is called the FilmEditor3D to correspond to the name of the Java class file which contains the main application. The requirements for the project subdivide into a number of distinct areas as shown below:

- Loading and saving of models and films.
- Interacting with films.
- Sequencing or editing films.
- Capturing films in other formats.
- Providing backgrounds.

These different areas are reflected in the package hierarchy of the FilmEditor3D which can be seen in Figure 1. More detailed descriptions of the packages, their contents and

![Figure 1: Package hierarchy of the FilmEditor3D.](http://www.java.sun.com/products/java-media/3D/1_3_api/)

interconnections are provided in Sections 2 and 3.

---

9Sun’s online tutorials can be found at http://www.java.sun.com/docs/books/tutorial/.

10Full specification at http://www.java.sun.com/products/java-media/3D/1_3_api/. 

2.1 FilmEditor3D

The FilmEditor3D class is the main class for the application and functions primarily as a user interface for the functionality that is provided by the different packages. It structures the layout of tools and options around the 3D canvas on which films will be displayed. It is actually a JApplet\textsuperscript{11}, but due to Java\textsuperscript{2} security restrictions it cannot actually run as an applet. Instead it uses a Java3D utility class called MainFrame\textsuperscript{12} which enables it to run as an application.

In addition to this the FilmEditor3D also controls which of the various controls and options are available depending on the current state of the system. In this guise it has three basic objectives:

- Monitor the status of the system and the progress of longer operations.
- Provide the user with useful feedback on what is happening.
- Enable or disable GUI components depending on whether the tools they represent should be available in the current state.

At first these goals seemed prohibitive to the distribution of functionality throughout the various packages, but this was overcome by use of PropertyChangeEvent\textsuperscript{13}. The FilmEditor3D implements the PropertyChangeListener\textsuperscript{14} interface, and all of the tools which have impact on the state of the application post PropertyChangeEvent which it detects. Upon receiving a PropertyChangeEvent the FilmEditor3D performs a selection of the following tasks based upon which property has changed:

- If the application has made a significant state transition, for example into the capturing film state, then the cursor is changed to provide a graphical signal to the user of what is happening.
- The status bar will usually be updated with a description of what has occurred (providing this is relevant to the user).
- If the property relates to the sequencing of the film then GUI components may be updated. For example, the average frame rate display is updated after each playback of a film sequence and the progress monitor is updated each time the frame being viewed changes.
- If a state is entered, or operation begun, which excludes other operations then they are disabled. This disabling has a visual effect on the GUI to inform the user that operations are not available.
- Similarly if a state is left, or operation finished, then any disabled operations are restored.

However the FilmEditor3D’s functionality is not entirely restricted to these user interface concerns, it also contains the code which creates and maintains the 3D universe in which

\textsuperscript{11}For details consult the Java API specification of javax.swing.JApplet.
\textsuperscript{12}For details consult the Java3D API specification of com.sun.j3d.utils.applet.MainFrame.
\textsuperscript{13}For details consult the Java API specification of java.beans.PropertyChangeEvent.
\textsuperscript{14}For details consult the Java API specification of java.beans.PropertyChangeListener.
the True 3D Films are viewed. This role consists of creating a SimpleUniverse\textsuperscript{15} and then attaching the components of the FilmEditor3D to its scene graph. Many of the components of the FilmEditor3D need to be attached in this way because they are created as specialized Java3D scene graph nodes with added functionality.

Default lighting is also added to facilitate viewing of models without textures. This lighting can be disabled by the user if it is not required.

2.2 Swing Issues

The javax.swing package is the easiest way to create a simple and easy to use GUI in Java\textsuperscript{2}. Unfortunately there are some unexpected interactions when using swing components with Java3D.

2.2.1 Lightweight Popups

The Canvas3D\textsuperscript{16} on which all 3D scenes are displayed is a heavyweight\textsuperscript{17} component whereas swing components are all lightweight\textsuperscript{18}. This means that popup menus and help tips do not get displayed properly. In a window which contains a Canvas3D, these popup items appear behind the canvas and are thus invisible to the user. The lightweight status of these popup items is however only a default and the desired behavior can be achieved by modifying their lightweight status in the following manner:

\begin{verbatim}
JPopupMenu.setDefaultLightWeightPopupEnabled(false);
ToolTipManager.sharedInstance().setLightWeightPopupEnabled(false);
\end{verbatim}

It would have been possible to use only heavyweight java.awt components instead, but they provide less friendly functionality, and also the lightweight popups are useful when capturing images from the canvas\textsuperscript{19}.

2.2.2 SwingWorker

Another swing problem arises when a significant amount of work is performed as the result of GUI interaction. In Java\textsuperscript{2} all graphical interactions are processed by the event dispatching thread. This means that if a button press, for example, starts a method which performs a large amount of work, then the event dispatching thread is held up doing this work. Unfortunately the event dispatching thread is also responsible for repainting all swing components. As a result, much of the GUI may flicker, disappear or fail to update while this thread is held up.

This is a well documented problem which has a simple solution. Sun provide a utility class SwingWorker\textsuperscript{20} that is available separately from the Java\textsuperscript{2} distribution and is intended to solve this problem. The SwingWorker operates by launching a new background thread\textsuperscript{21} to perform whatever processing is necessary whilst freeing up the event dispatching thread to continue its work. For convenience the SwingWorker class has been included as part of the fe3d.util package within this project.

\textsuperscript{15}For details consult the Java3D API specification of com.sun.j3d.utils.universe.SimpleUniverse.
\textsuperscript{16}For details consult the Java3D API specification of javax.media.j3d.Canvas3D.
\textsuperscript{17}Heavyweight popups use native system resources and interfere with lightweight components.
\textsuperscript{18}Lightweight components are more efficient but only work well without heavyweight components.
\textsuperscript{19}See the description of the fe3d.capture package for more details on why lightweight popups are useful.
\textsuperscript{20}For SwingWorker details see http://java.sun.com/products/jfc/tsc/articles/threads/update.html.
\textsuperscript{21}For swing thread issues see http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html.
2.3 Performance Constraints

Displaying True 3D Films at realistic frame rates makes serious demands of system resources. However there are three particularly significant factors which affect the ability of the FilmEditor3D to deliver acceptable performance.

2.3.1 Rendering Speed

Perhaps most obviously the system needs to be able to render frames fast enough to display the film. The models generated as part of a film may contain tens of thousands of polygons\(^ {22}\), each of which may need to be rendered every frame. Adding textures and lighting to a scene also adds complexity to the rendering, and in the case of True 3D Films, where textures are used very heavily, this has a significant impact on the range of frame rates which are feasible.

Thankfully the Java3D API makes use of many of the available hardware optimizations and in most cases a good graphics card will be enough to handle these rendering requirements. However, to ensure good performance with longer films, a graphics card with a large amount of dedicated memory should be used, because the time taken in transferring models between system RAM and the graphics card’s memory can cause a reduction in the achievable frame rate\(^ {23}\).

2.3.2 Modelling Updates

Modelling updates affect frame rates in two different ways:

- Changes to the scene being viewed mean that new data has to be passed to the hardware. This slows rendering, particularly if there are many optimizations to be performed on the new data before displaying it.

- Actually changing the scene takes time. These updates are performed by the system processor not optimized graphics hardware, so large updates may incur significant time penalties on slower systems.

Modelling updates within the FilmEditor3D do not occur in the way that might be expected. The geometry of the surfaces being displayed is never changed because each deformation of the model is actually a separate model with its own associated geometry. This means that once the geometry has been optimized for the first time it never needs to be optimized again\(^ {24}\).

Therefore modelling updates are most commonly just a selection of which model, and hence which piece of geometry, should be visible. These selections correspond to relatively simple adjustments to the scene graph which are made very efficiently by Java3D\(^ {25}\).

However, there are other modelling updates used within the FilmEditor3D which can have a more noticeable impact on performance. These are frame by frame changes to rendering attributes and viewing transformations. In order to provide interesting visualizations the FilmEditor3D allows the user to change camera positions, or alter the way in which shapes and surfaces are rendered\(^ {26}\).

\(^{22}\) Some of the models used in testing the FilmEditor3D had over 20,000 polygonal faces.

\(^{23}\) The FilmEditor3D was tested and performed well using a 64MB GeForceII Pro graphics card.

\(^{24}\) Refers to optimization performed before data goes to graphics card e.g. geometry stripping.

\(^{25}\) Details of how this model selection is accomplished are in the description of the fe3d.sequence package.

\(^{26}\) Details of how attributes are modified are in the description of the fe3d.sequence package.
These changes necessitate modification of various attributes within the scene graph, and a fast system is needed to ensure that these modifications do not hold up the graphics hardware. One additional issue is that if updates are not generated on a per frame basis then those frames requiring updates may take longer to render than others. For this reason the FilmEditor3D requires a reasonably fast system processor in order to deliver acceptable performance.  

2.3.3 Texture Usage

The final and potentially most significant factor that can affect the applications performance is texture usage. By their nature True 3D Films are designed to be photorealistic and this means they make heavy use of detailed textures mapped onto the surfaces within a scene. This may not appear to be a problem but it creates two major concerns:

- Rendering of textured objects is almost always slower than rendering pure geometry so texturing every surface in a scene has a noticeable impact on frame rates.
- Textures themselves use up memory, particularly when they are large. Keeping all of the necessary textures available requires a large amount of RAM, otherwise they can be paged out to the hard disk. Recovering textures from disk is a significant overhead and results in very poor performance.

To see how this latter case is significant a small example based upon one of the films used in testing is illustrated below:

Based on 4 RGB textures per model and 1s of film at 25 frames/sec:

<table>
<thead>
<tr>
<th>Description</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>texture size</td>
<td>no. of pixels * no. of bits per pixel</td>
</tr>
<tr>
<td></td>
<td>= (640 * 480) * 24</td>
</tr>
<tr>
<td></td>
<td>= 7,372,800 bits</td>
</tr>
<tr>
<td></td>
<td>= 900K</td>
</tr>
<tr>
<td>model size (textures only)</td>
<td>no. of textures * texture size</td>
</tr>
<tr>
<td></td>
<td>= 4 * 900</td>
</tr>
<tr>
<td></td>
<td>= 3,600K</td>
</tr>
<tr>
<td>film size (textures only)</td>
<td>no. of models * model size</td>
</tr>
<tr>
<td></td>
<td>= 25 * 3600</td>
</tr>
<tr>
<td></td>
<td>= 90,000K</td>
</tr>
<tr>
<td></td>
<td>~ 88MB</td>
</tr>
</tbody>
</table>

That is roughly 88MB of raw texture data for one second of film. Actual values are often worse than this projection because Java3D requires textures to have dimensions which are powers of two, and often this requires that the TextureLoader perform some resizing of the texture images. It is not uncommon for resizing to result in an entire duplicate

\footnotetext[27]{Testing indicates that a ~1GHz or faster processor delivers satisfactory performance in this respect.}

\footnotetext[28]{This is obviously hardware dependent but true in general.}

\footnotetext[29]{For details consult the Java3D API specification of javax.media.j3d.Texture2D.}

\footnotetext[30]{For details consult the Java3D API specification of com.sun.j3d.utils.image.TextureLoader.}
copy of the image being held in memory, which doubles the amount of memory being consumed by texture data\textsuperscript{31}.

This additional memory consumption makes it very desirable to resize images before they are passed to the TextureLoader. The \texttt{fe3d.io.loaders.vrml1} package contains an additional class \texttt{Texture2(new)}\textsuperscript{32} which bypasses the TextureLoader completely and resizes images itself. However this class is not currently in use because it slows down loading significantly and still generates extra images, although these images are made available for garbage collection more quickly than those created by the TextureLoader. The best results are achieved by resizing all texture images outside of the FilmEditor3D preventing any extra image generation from taking place.

Unfortunately even taking such preemptive measures does not eliminate the texture problem. Transfer times between RAM and the graphics card are normally fast enough to maintain a good frame rate for textured surfaces, provided all of the textures can be held in RAM simultaneously. However, as soon as there is too much texture data to fit in RAM at one time, frame rates drop drastically as data is transferred to and from disk. Therefore to ensure good performance the system must have enough memory to hold the entire film in RAM\textsuperscript{33}.

\section{Package Overview}

The \textit{FilmEditor3D} is comprised of a number of packages as indicated in Section 2. The details of how these packages depend on one another can be seen in Figure 2. In addition

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{dependency_diagram.png}
\caption{Package dependencies in the \textit{FilmEditor3D}.}
\end{figure}

\textsuperscript{31}For one film used during testing this resulted in 14MB of texture related data per frame.

\textsuperscript{32}To use this class rename it to \texttt{Texture2} and remove the original \texttt{Texture2} class.

\textsuperscript{33}Most films tested required only \textasciitilde512MB of RAM but 1 or 2GB would be preferable.
to the way they interconnect. These packages also have a common high level structure. Each package contains a number of classes which provide its functionality. It then contains a sub-package called action. These action packages contain a number of classes, each of which extend the Java2 class AbstractAction. These actions can be used to create GUI components such as buttons or menu items that allow graphical interaction with the classes from the enclosing packages.

So, for example, the actions contained in the fe3d.capture.action package can be used to create graphical controls for the classes in the fe3d.capture package. This greatly simplifies the creation of a GUI which can control these disparate components because each package defines its own GUI controls which can easily be included in a unified user interface. It also allows an application to provide the user with a high level of functionality without requiring in depth knowledge of how each package is implemented.

4 Vrml1 Package

The models generated by the dynamic capture system within the department are currently all VRML Version 1.0. Unfortunately none of the currently available VRML loaders for Java3D support this specification because it has been replaced by the newer VRML97 format. Conversion tools are freely available but are platform dependent and are thus unsuitable for use in a generalized tool. To overcome this obstacle the fe3d.io.loaders.vrml1 package has been created for loading these old specification VRML models into Java3D.

4.1 Parser

The Parser class is a general parser for the VRML1.0 format. The parser is a complete implementation with methods that will recognize and return all nodes, fields and values which are valid for the VRML1.0 specification. The Parser has an internal Hashtable which links the string representations for VRML nodes (or fields) to corresponding integer flags.

When asked for the next node, field or item the parser reads from the input file and looks up the Hashtable before returning the appropriate flag. These flags are accessible to all classes within the fe3d.io.loaders.vrml1 package. This means that the parser can easily be used to identify nodes (or fields) as shown in the example below:

```java
int nodeType = parser.nextNode();
if (nodeType == Parser.CUBE) ... perform code for handling cube
```

The VRML1.0 format also contains a number of different types of grouping nodes. These grouping nodes may contain either other nodes, fields or both. This makes it impossible to determine in advance whether the next item in the input stream will be a node or a field when parsing one of these groups. To handle this the parser has methods that get the first and next items, where an item may be either a node or a field.

These methods again return an integer flag but in this case if the item read was a field then the flag includes an offset. The offset is necessary because there is an overlap between the values of the flags used to represent nodes and those which represent fields.

---

34 For details consult the Java2 API specification of javax.swing.AbstractAction.
35 Graphical User Interface.
36 Full specification of this format can be found at http://www.web3d.org/VRML1.0/vrml10c.html
37 The overlap is because both sets of flags are used to index arrays and therefore must start at 0.
The type of the item can be identified by checking for the presence of the offset as shown below:

```java
int itemType = parser.nextItem();

// No offset, itemType is a node flag
if (itemType < Parser.IS_FIELD)
    handleNode(itemType);

// Offset, subtract offset from itemType to get a field flag
else
    handleField(itemType - Parser.IS_FIELD);
```

The `Parser` also has methods to retrieve instances of each of the different data types which may be present in a VRML1.0 file. When asked for an instance of a particular data type it reads a value (or values in the case of multi-valued types) and attempts to convert it to the specified type. If successful this value is returned, if not an exception is thrown.

### 4.2 LoaderState

The VRML1.0 format uses the concept of a loader “state” which contains attributes of the modelling environment which may be referred to whilst construction of a 3D model or scene is underway. The `LoaderState` class corresponds to this abstract state and contains information about the following attributes:

- Material properties: Including ambient, diffuse, specular and emissive colours.
- Geometric properties: Including coordinates and face sets.
- Normal properties: Including normal vectors or a crease angle for automatic generation of surface normals.
- Texture properties: Including texture images, texture coordinates and boundary modes.

In addition to methods for setting and retrieving all of these various attributes the `LoaderState` also has a method `copyInto(LoaderState state)` that is used to duplicate the entire state. This is necessary because the scope of the state is limited to the enclosing group, and so the state has to be saved and restored whenever traversing a group construct.

### 4.3 V1Loader

The `V1Loader` class actually performs the work of loading the VRML models. It implements the `Loader`\(^{38}\) interface so that it can be reused as a generic model loader for Java3D. This means that all loading is accomplished by means of three very simple steps:

1. Construct a new loader `loader = new V1Loader();`.

\(^{38}\)For details consult the Java3D API specification of `com.sun.j3d.loaders.Loader`. 
2. Set the base path for texture images etc. `loader.setBasePath(basePath);`

3. Load the scene `Scene scene = loader.load(file);`

However the VRML1.0 specification is quite large and although the `Parser` can recognise all valid VRML1.0 nodes, the loader only supports the subset of these necessary for loading a *True 3D Film*. The currently supported nodes are:

- Material properties: `Material`
- Geometric primitives: `Cone`, `Cube`, `Cylinder`, `Sphere`
- Indexed geometry: `Coordinate3`, `IndexedFaceSet`
- Surface properties: `Normal`, `ShapeHints`
- Light sources: `DirectionalLight`
- Transformations: `MatrixTransform`, `Rotation`, `Scale`, `Transform`, `Translation`
- Texture properties: `Texture2`, `TextureCoordinate2`
- Grouping nodes: `Separator`

In addition to those above the following nodes, although not supported, are safely ignored by the loader with a message being displayed on standard output:

- `AsciiText`
- `FontStyle`
- `Info`
- `MaterialBinding`

The `V1Loader` works by repeatedly retrieving the next node from the `Parser` and identifying it. Once a node is identified it is farmed out to the corresponding class to be dealt with. There are separate classes for each of the supported node types. These classes retrieve the necessary fields and values from the `Parser` and make changes to the `LoaderState` or create new scene graph nodes as appropriate. The only exception to this is the `Separator` node which affects the way loading proceeds, and so is handled within the loader class.

Adding support for currently unsupported node types is quite simple. The `Parser` will identify them correctly, so all that is necessary is to create a method or class to handle the new node type, and then amend the `handleNode()` method in the `V1Loader` class to call this new code when such a node is encountered.

## 5 Loaders Package

Java3D provides a `Loader`\(^{39}\) interface for classes which wish to load models into a Java3D scene graph. There are many freely available loaders for Java3D supporting a variety of different formats\(^{40}\). For this reason the `fe3d.io.loaders` package contains a `LoaderControl` class to handle all loading of external file formats using `Java3D Loaders`.

\(^{39}\)For details consult the Java3D API specification of `com.sun.j3d.loaders.Loader`.

\(^{40}\)For a full list of currently available loaders consult [http://www.j3d.org/utilities/loaders.html](http://www.j3d.org/utilities/loaders.html)
5.1 LoaderControl

The LoaderControl provides a simple file chooser and then identifies the correct Loader to use based upon the type of the selected file(s). It also provides elementary progress monitoring throughout the loading operation. Currently the LoaderControl only supports two file formats:

- The Wavefront .obj format. For which the Loader is provided as part of the Java3D distribution.
- The VRML1.0 .wrl format. Supported by the V1Loader which forms part of the FilmEditor3D implementation.

Additional Loaders can be registered with the LoaderControl in two steps. Firstly the Loader has to be registered with the file chooser so that it can detect files of the new format. This is accomplished by adding the following two lines into the LoaderControl constructor:

```java
fileFilter = new LoaderFileFilter(mySuffix, myDescription);
fileChooser.addChoosableFileFilter(fileFilter);
```

Secondly a line has to be added that will instantiate the proper Loader if a file of the new format is encountered. This is accomplished by adding the following line to the LoaderControl code:

```java
else if (file.getName().indexOf(mySuffix) != -1) loader = new myLoader();
```

The above line should be added immediately following these two lines:

```java
if (file.getName().indexOf(".wrl") != -1) loader = new V1Loader();
else if (file.getName().indexOf(".obj") != -1) loader = new ObjectFile();
```

The fe3d.sequence.Sequencer and fe3d.background.BackgroundGroup both require the filenames for each model they are given. For this reason the LoaderControl returns a SceneBase\(^{41}\) from which all loaded models may be retrieved along with their names by invoking the getNameObjects() method.

Unfortunately this approach disregards the ordering of the models. This ordering is important because the models have often been selected in the correct order for a film. To overcome this problem an ordered list of the model names, separated by carriage returns, is placed in the SceneBase’s description. This means that the following code can be used to retrieve models in the correct order:

```java
String modelList = scene.getDescription();
Hashtable models = scene.getNameObjects();
int index = 0;
while ((index = modelList.indexOf("\n")) != -1)
{
    String name = modelList.substring(0, index);
    modelList = modelList.substring(index + 1);
    Node model = models.get(name);
}
```

\(^{41}\)For details consult the Java3D API specification of com.sun.j3d.loaders.SceneBase.
5.2 Import Actions

There are two actions which make direct use of the `LoaderControl` class. They are the `fe3d.io.action.ImportFrameAction` and the `fe3d.io.action.ImportBackgroundAction`. They are both straightforward and use the `SwingWorker` class to perform the loading in the background so as not to disrupt user interactions. The only difference between the two is that the `ImportFrameAction` adds the loaded models into the `fe3d.sequence.Sequencer` whereas the `ImportBackgroundAction` adds them to the `fe3d.background.BackgroundGroup`.

6 IO Package

The `fe3d.io` package provides the means to load and save 3D films which have been generated by the `FilmEditor3D`. These films are stored in a custom ASCII format which was designed so that users can edit film files manually if they wish. The `io` package has three major components, the `FilmWriter`, `FilmReader` and `FilmParser`.

There are also two associated `AbstractActions` the `fe3d.io.action.OpenAction` and the `fe3d.io.action.SaveAsAction`. These actions simply invoke `FilmReader` and `FilmWriter` methods from within a `SwingWorker` so as to prevent the GUI difficulties described previously.

6.1 FilmWriter

The `FilmWriter` retrieves the settings for the `Player` along with the contents of the `Sequencer` and `BackgroundGroup`. This data is then written to a film file in the following form:

1. A `FilmState` block is output containing:
   
   (a) The `Player` timer delay.
   
   (b) The `Player` loopCount.
   
   (c) The `Player` sequence startIndex.
   
   (d) The `Player` sequence endIndex.

2. A `BackgroundStore` block is output containing:

   (a) A `Background` block for each `BackgroundModel` which contains:
   
       i. The `BackgroundModel` filename.
   
       ii. The `BackgroundModel` visible flag.

3. A `ModelStore` block is output containing:

   (a) A `Model` block for each `FrameModel` which contains:
   
       i. The `FrameModel` filename.

4. A `PlaySequence` block is output containing:

   (a) A `Frame` block for each `SequencerFrame` which contains:
   
       i. The `SequencerFrame` filename.
   
       ii. The `SequencerFrame` textureMode attribute.
iii. The `SequencerFrame` `polygonMode` attribute.
iv. The `SequencerFrame` `shadeModel` attribute.
v. The `SequencerFrame` `cameraTransform` viewing transformation.

6.2 FilmReader

The `FilmReader` uses the `FilmParser` to parse in a `FilmEditor3D` film file. As `BackgroundModels` and `FrameModels` are encountered a `LoaderControl` object is used to import them. Since the film format was created with manual editing in mind, the `FilmReader` is designed to handle the different blocks in any order. Also any or all of the blocks or fields may be omitted, default values will be substituted where necessary.

7 Background Package

The `fe3d.background` package is intended to group together all of the classes needed to provide the functionality for adding static\(^{42}\) background environments to a film. This is a relatively simple package and its structure can be seen in Figure 3. The classes themselves are also relatively straightforward.

7.1 BackgroundModel

The `BackgroundModel` class is simply a `BranchGroup`\(^{43}\) with three additional features:

- The `ALLOW_DETACH` capability is set to enable it to be added to or removed from a live scene graph\(^ {44}\).
- It has a boolean flag to indicate whether the background it contains should currently be visible or not.
- The name of the file from which the model was obtained is stored with the model.

These additions are to allow different backgrounds to be made visible. The flag indicates whether or not the background stored in the `BackgroundModel` should be visible, the `ALLOW_DETACH` capability makes it possible to remove any non-visible backgrounds from the scene graph and the filename is used as a handle for selecting between different backgrounds. The filename is also important for IO purposes as a list of the backgrounds in use forms part of a saved film.

The `BackgroundModel` class also has elementary methods for querying these properties and the ability to set the visibility flag to a given value. It should be noted that setting the visibility flag does not explicitly change a background’s status, it simply signals to any code that manipulates these backgrounds that the background should be added to or removed from the scene graph as appropriate. In the `FilmEditor3D` this control is provided by the `BackgroundGroup` class.

\(^{42}\)Backgrounds could include animation, static only indicates they are not sequenced as part of a film.

\(^{43}\)For details consult the Java3D API specification of `javax.media.j3d.BranchGroup`.

\(^{44}\)A scene graph is live if it is attached to a `Universe`.

7.2 BackgroundGroup

The BackgroundGroup class is slightly more complicated but is again an extended BranchGroup. It contains a model store which is a java.util.Hashtable that holds references to all of the currently loaded BackgroundModels, indexed by their filenames. It also maintains two lists:

- backgroundModelList: A list containing the names of all the currently available models.
- visibleBackgroundsList: A list containing the names of all of the above models that are currently visible.

The BackgroundGroup has an on-screen dialog that allows the user to move backgrounds between these two lists, and hence select which of the available backgrounds should be displayed. As the user moves backgrounds to or from the visible list the corresponding BackgroundModels are added to or removed from the scene graph. The models are added as children of the BackgroundGroup which controls them, so that no knowledge of how the scene graph has been constructed is required.

This makes using a BackgroundGroup very easy. It can be treated as a normal scene graph node by Java3D and, provided it has been added to the scene graph, any addition
or selection of backgrounds takes place behind the scenes. This relieves user code of
the need to keep track of where backgrounds are in the scene graph. This also offers
the additional possibility of adding multiple BackgroundGroups to the same scene graph
but in different locations. This would allow localized control over which elements of the
background were visible.

8 Sequence Package

The fe3d.sequence package contains a large proportion of the functionality of the FilmEditor3D. It provides all of the sequencing and playback features as well as incorporating
control over rendering styles and preset camera positions. The structure of the package
reflects this diversity and is quite complex\(^{45}\), as can be seen Figure 4.

\[\text{Runnable} \quad \text{java.lang} \quad \text{BranchGroup} \quad \text{Transform3D} \quad \text{fe3d.sequence} \]

\[\text{FrameModel} \quad \text{SequencerFrame} \quad \text{SequencerProperties} \quad \text{Player} \quad \text{Sequencer} \]

\[\text{javax.media.j3d} \quad \text{javax.media.j3d} \quad \text{FilmEditor3D packages} \quad \text{Java2/Java3D packages} \]

\[\text{TransformGroup} \quad \text{Switch} \quad \text{Behavior} \quad \text{WakeUpCriterion} \quad \text{java.lang} \]

\[<<\text{interface>>} \quad \text{Runnable} \]

Figure 4: Internal class structure of the fe3d.sequence package.

8.1 FrameModel

For a True 3D Film every frame is in fact a 3D model. Hence the FrameModel class
is intended to encapsulate one frame from a 3D film. The FrameModel is actually a
SharedGroup\(^ {46}\) with two additional features:

- The name of the file from which the model was obtained is stored with the model.
- When the model is created, it is searched and all of the Appearance node components
  which it contains are stored in a list.

\(^{45}\) The actions from the fe3d.sequence.action package have been omitted to simplify the diagram.

\(^{46}\) For details consult the Java3D API specification of javax.media.j3d.SharedGroup.
The file name is stored with the FrameModel so that it can be used as a handle for the model. This handle will be used to identify the model for sequencing purposes and is also necessary to enable the model to be saved as part of a True 3D Film.

Ordinarily changing the appearance of a model would require a traversal of the scene graph to find the relevant Appearance node components and modify their values. Performing this traversal in advance and storing a reference to each of the relevant Appearances makes it possible to alter the appearance of the model without repeated, time consuming traversals of the scene graph. Furthermore, because the traversal is performed before the model has been compiled or made part of a live scene graph, no capability bits need to be modified in order to make traversal possible.

The reason that the FrameModel is a SharedGroup is simple. Each model represents one frame of a 3D film, in the process of sequencing a film it is only natural that the same model may be used more than once. If the whole model were to be duplicated a significant wastage of resources would be incurred. By having each model as a SharedGroup it can be referenced multiple times by different Link nodes, without the model itself having to be duplicated. For films with large models or a high level of frame reuse the savings in time and space can be quite significant.

### 8.2 SequencerFrame

Although the FrameModel class holds a single 3D frame in a manner suitable for reuse, the Sequencer has more demanding requirements for a model that is to form part of a play sequence. That is why the SequencerFrame class was created. A SequencerFrame is a specialised BranchGroup that contains a Link to a FrameModel. The SequencerFrame has the following additional features:

- The ALLOW_DETACH capability is set to enable it to be added to or removed from a live scene graph.
- Texture, render style and shading style attributes can be stored with each frame.
- A viewing transformation can be stored with each frame.

The ALLOW_DETACH flag enables a frame to be added to or removed from a live scene graph depending on whether it forms part of the current play sequence or not. The storage of attributes along with each frame means that individual frames can have different rendering attributes, or can be viewed from different positions. Also, because each SequencerFrame contains a Link to a FrameModel, it is possible for the same model to be linked to by several different frames, each of which can render this same model in a different way, or from a different position, based upon the properties stored with the frame.

As a simple example of this, consider a play sequence of 360 SequencerFrames, each of which has a link to the same FrameModel which is centred at the origin. Now assume each frame has a stored viewing transformation that corresponds to a rotation about the origin through an angle which increases by 1 degree between subsequent frames. It is not too difficult to deduce that when played, this sequence will appear to rotate once around the model. The important fact to note, however, is that rather than using 360 different models, each a transformation of the others, only 1 model is used, a much more efficient use of resources.

\footnote{For details consult the Java3D API specification of }
8.3 Sequencer

The Sequencer class is quite complex but has a very similar underlying structure to the fe3d.background.BackgroundGroup. The Sequencer is an extended Switch\footnote{For details consult the Java3D API specification of javax.media.j3d.Switch.} group. A Switch group is ideal for animation because it maintains the ordering of its children, and allows one of its children to be selected for display at a time. The Sequencer makes use of these properties when animating a play sequence, but in order to provide proper sequencing capabilities it has to have some additional functionality.

8.3.1 Play Sequence Control

The Sequencer contains a model store which is a java.util.Hashtable that holds references to all of the currently loaded FrameModels, indexed by their filenames. It also maintains two lists:

- \textit{frameModelList} : A list containing the names of all the currently available models.
- \textit{sequencerFrameList} : A list containing the names of all of the above models that are currently linked to by frames in the play sequence.

The Sequencer has an on-screen dialog that allows the user to move models between these two lists, and hence select which of the available models should form part of the play sequence. This dialog also allows the same model to be added to the play sequence multiple times and provides the user with controls for adjusting the position of a model within the play sequence.

As the user adds a model to the frame list a new SequencerFrame is created to encapsulate it as a frame in the play sequence. Similarly when the user removes a frame from the play sequence the appropriate SequencerFrame is discarded. These frames are added to or removed from the scene graph as children of the Sequencer. This means that playing the sequence is simply a matter of stepping through the children of a Switch node.

However the number of SequencerFrames created and discarded using this approach can be quite large. Having a significant number of these discarded frames waiting for the garbage collector can waste a lot of resources, particularly when new frames also need to be created. To overcome this the Sequencer maintains a set of “free” SequencerFrames. As frames are discarded they are added to this set and labelled as “free”. Requests for new frames first check if there are any “free” frames available, if there are then they are reused. Only if all of these frames have been reclaimed will new SequencerFrames be created.

8.3.2 Appearance Attributes

The Sequencer is also responsible for modifying the attributes that govern the appearance of models within the play sequence. There are three attributes which can be modified:

- \textit{RenderingAttributes}\footnote{For details consult the Java3D API specification of javax.media.j3d.RenderingAttributes.} : Control whether shapes are drawn as vertices, lines, or filled polygons.
- \textit{ColoringAttributes}\footnote{For details consult the Java3D API specification of javax.media.j3d.ColorringAttributes.} : Control whether shapes are shaded using flat or Gouraud shading.
- TextureAttributes\textsuperscript{51} : Control the way in which textures are applied to surfaces. Changes to these attributes can come from two places:
  - The user can modify attributes or
  - The current frame may have stored attributes.

In either case the system for updating the models is the same. The fact that a FrameModel maintains a list of the Appearances which it contains eliminates the need for a costly scene graph traversal, but working through all of the attributes for all of the Appearances would still use up a lot of CPU time. Instead the Sequencer uses attribute sharing to speed up this process.

As part of its instantiation the Sequencer creates one instance of each of the three attribute types named above. It then sets the necessary capability bits to allow these attributes to be updated when they become live, for example:

```java
    colourAttr = new ColoringAttributes();
    colourAttr.setCapability(ColoringAttributes.ALLOW_SHADE_MODEL_READ);
    colourAttr.setCapability(ColoringAttributes.ALLOW_SHADE_MODEL_WRITE);
```

Then whenever a FrameModel is added to the model store, the Sequencer iterates through the model’s list of Appearances, setting each to have the attributes which the Sequencer has created. This means that every model takes its attribute information from one shared source. To make a change to the attributes of all the models, the Sequencer simply modifies the appropriate shared attributes object. This sharing makes modifying attributes much faster and eliminates many unnecessary attributes objects. This in turn frees up more system resources for other operations.

### 8.3.3 Viewing Transformations

The final duty that the Sequencer performs is to update the view transform whenever a SequencerFrame has a stored viewing transformation. The Sequencer has a reference to the Universe’s view platform TransformGroup\textsuperscript{52}. The Sequencer simply updates this TransformGroup with the stored transformation from the SequencerFrame.

### 8.4 Player

The Player class is responsible for actually controlling playback of a play sequence. It maintains a reference to the Sequencer and selects which frame should be displayed at a given point using the setWhichFrame(int index) method. It provides a number of common controls over playback which operate in a standard manner. These controls are:

- Step backwards one frame.
- Step forwards one frame.
- Jump to a specific frame.
- Play the play sequence.

\textsuperscript{51}For details consult the Java3D API specification of javax.media.j3d.TextureAttributes.
\textsuperscript{52}For details consult the Java3D API specification of javax.media.TransformGroup.
8.4.1 Timing

Ensuring that the timing was correct during playback was one of the biggest challenges in the project. The first and simplest attempt was to use a `SwitchValueInterpolator` which would perform a linear interpolation through the play sequence. This approach suffered two serious faults:

- There was no way of pausing and resuming playback.
- The amount of work done by the `Sequencer` and `Java3D` renderer between frames is quite high. The interpolator guarantees that it will complete the interpolation in a specified time. Unfortunately with complex models the renderer was taking longer than the interpolator had allocated for each frame and so the interpolator skipped intermediate frames to compensate.

The second approach overcame both of these problems. The `Player` was redefined as an extended `Behavior` with a customised wakeup condition. The `Player` used a `WakeupOnElapsedTime` criterion to wake it when the next frame should be displayed. The elapsed time was calculated to fit the desired frame rate. Upon waking the `Player` moved to the next frame in the sequence, reset the wakeup condition and then went back to sleep.

This approach offered the ability to closely control the progress through the play sequence and never skipped frames. However, once implemented it became apparent that there was a significant time overhead associated with the use of the `WakeupOnElapsedTime` condition. Using this condition the `Player` was never woken more frequently than once every 60 milliseconds (Even with an elapsed time of 0). This restricted the maximum frame rate to roughly 15 frames per second, which was not nearly fast enough for interactive visualization.

The third and final approach is by far the most successful. The convenience of having the `Player` as a `Behavior` has been kept but the `WakeupOnElapsedFrames` condition has been used instead. Using this criterion with an argument of 0 causes the `Player` to be woken after every frame is rendered. This immediately increased the frame rate to about 40 frames per second as there is almost no overhead with this wakeup condition, but a new method for controlling the frame rate had to be devised.

The solution was to employ a separate thread to act as a dedicated timer. The timer simply sleeps for a specified number of milliseconds before terminating. Every time the `Player` is woken up it performs its work and then waits for the timer thread to terminate. Once the timer terminates the `Player` simply launches a new timer and goes back to sleep.

Since a new timer is launched immediately upon the termination of the previous one, the time between subsequent timer threads being launched, and hence the time between the `Player` being woken, remains constant. So adjusting the length of the timer’s sleep

---

53 For details consult the Java3D API specification of `javax.media.j3d.SwitchValueInterpolator`.
54 For details consult the Java3D API specification of `javax.media.j3d.Behavior`.
55 For details consult the Java3D API specification of `javax.media.j3d.WakeupOnElapsedTime`.
56 For details consult the Java3D API specification of `javax.media.j3d.WakeupOnElapsedFrames`. 
corresponds to adjusting the frame rate. However, the overhead associated with launching a new thread is platform dependant so frame rates cannot be specified as absolute values. Instead the timer delay can be adjusted by the user. After playback the achieved frame rate is calculated and using this figure the delay can be adjusted until a reasonable approximation of the desired frame rate has been achieved.

8.4.2 SettingsDialog

The Player provides a simple settings dialog which allows the user to modify the following properties:

- The start and end indices of the play sequence can be changed to allow play back of subsets of the whole sequence.
- The loop count can be changed so that the Player loops through the sequence a specified number of times.
- The timer delay can be changed to achieve different frame rates.

8.5 Sequence Actions

There are a large number of AbstractActions in the fe3d.interaction.action package. They are all quite trivial and simply provide a GUI wrapper for the methods and classes discussed above.

9 Interaction Package

The fe3d.interaction package groups together all of the classes needed to provide the functionality for interactively controlling the position of the camera whilst viewing a 3D film. Once again the package is relatively simple, its structure can be seen in Figure 5. The majority of the work done in this package is performed by the OrbitBehavior class.

9.1 OrbitBehavior

The OrbitBehavior class is actually provided as part of the Java3D distribution but due to several bugs in the code the distributed version is not suitable for use in the FilmEditor3D. The version used in the fe3d.interaction package was provided by the Java3D development team at Sun and is used with their permission. This corrected class will be present in the next beta release of java3D. Once this new release of Java3D is being used the following changes need to be made to the FilmEditor3D to ensure that the most up to date version of the OrbitBehavior class is being used:

- The OrbitBehavior class should be deleted from the fe3d.interaction package.
- The line “import fe3d.interaction.OrbitBehavior;” should be replaced by the line “import com.sun.j3d.utils.behaviors.vp.OrbitBehavior;” in the following classes:

\[57\] Dashed occurrence of the class in Figure 5 indicates the OrbitBehavior’s original location.

\[58\] First release containing the corrected class will be j3d1.3beta2.
The OrbitBehavior enables the user to zoom, translate and rotate about a point of interest within a 3D scene using the mouse\textsuperscript{59}. By default the position of interest is the origin and since all of the models loaded by the FilmEditor3D are centred at the origin, this default is left unchanged. The OrbitBehavior also provides the ability to move the camera to a predetermined “home” position which has been specified by the user. This is particularly useful if the user becomes lost in a potentially infinite universe.

\textsuperscript{59}For details consult the Java3D API specification of \texttt{com.sun.j3d.utils.behaviors vp.OrbitBehavior}.
9.2 OrbitModifier

The OrbitModifier is a JFrame\textsuperscript{60} that contains graphical controls for modifying the properties of an OrbitBehavior. The modifier allows adjustments to be made to the following aspects of the OrbitBehavior so that the user can customize the feel of the camera interactions to suit their needs:

- Sensitivity of translation along x and y axes can be changed.
- Sensitivity of rotation around x and y axes can be changed.
- Sensitivity of zoom along the z axis can be changed.
- Translation, rotation and zoom controls can have their orientation reversed.
- Translation, rotation and zoom interactions can all be enabled or disabled individually.
- Zoom can be made proportional to distance from the point of interest rather than absolute.

The OrbitModifier applies all these changes to a test behavior which operates on a small test canvas containing a cube. This enables the user to try out the various settings before they are applied to the real OrbitBehavior, and prevents unintended modifications from affecting established interaction techniques.

9.3 Actions

The fe3d.interaction.action package contains four AbstractActions to provide user interface components that can be integrated into the GUI of an application:

- The GoHomeAction uses the OrbitBehavior to move the camera to its home position.
- The SetHomeAction sets the current camera position to be the new home position for the camera.
- The RestoreHomeAction resets the camera home position to its original value.
- The ShowCameraControlsAction causes the OrbitModifier to display its GUI controls on screen.

Each of these classes has a very straightforward implementation, simply invoking the relevant methods from the OrbitBehavior or OrbitModifier class as appropriate.

10 Capture Package

One of the most important features of the FilmEditor3D is the ability to capture 2D visualizations in different ways. This functionality is provided by the fe3d.capture package. The package provides means to obtain three different types of output:

\textsuperscript{60}For details consult the Java API specification of javax.swing.JFrame.
• A single frame captured as a JPEG image.
• A film sequence captured as a series of numbered JPEG images.
• A film sequence captured as a conventional 2D film.

All three of the above capture types rely on the use of the Raster class to extract an Image from the frame buffer. In order to accomplish the last method of capture the package also utilises classes and functionality provided in the Java Media Framework API. This means that the fe3d.capture package uses classes from the Java2, Java3D and JMF APIs. This is reflected in the complex internal structure of the package as shown in Figure 6.

![Figure 6: Internal class structure of the fe3d.capture package.](image)

### 10.1 CaptureDialog

The CaptureDialog class is a customized JDialog which allows the user to control settings that affect the capture of images and films. These options enable the user to:

• Control the quality of the still images being captured. This is done by modifying the quality parameter of the JPEGImageEncoder.

---

61Currenty the package generates output in either the windows .avi or quicktime .mov format.
62For details consult the Java2 API specification of javax.media.j3d.Raster.
63For details consult the Java2 API specification of java.awt.Image.
64For full details of the JMF API consult http://www.java.sun.com/products/java-media/jmf/.
65For full details consult the Java2 API specification of javax.swing.JDialog.
66For full details consult the Java2 API specification of com.sun.image.codec.jpeg.JPEGImageEncoder.
• Control the frame rate of generated films. Frame rate specified is passed to the `CaptureCanvas3D` which determines the media format accordingly.

• Select destination files for both still and live capture data. In the case of live capture, the filetype of the destination determines the format used for output. So, for example, specifying a destination of “Film.jpg” results in a sequence of JPEG images being generated, whereas “Film.mov” would result in the creation of a movie file.

The `CaptureDialog` does not actually take any action based upon these settings. Instead it provides methods for querying each of the properties, so that when capture is initiated the settings can be retrieved by the capture code. The `CaptureDialog` has an associated `AbstractAction`, the `ShowCaptureDialogAction` which makes it easy to request the dialog be displayed, from within the GUI of the `FilmEditor3D`.

### 10.2 ImageSourceStream and ImageDataSource

`JMF` has a very simple approach to handling media. It operates on the principle that all media comes from a data source, is processed, and goes to a data sink. In the `fe3d.capture` package the data source is the `CaptureCanvas3D` and the data sink corresponds to the destination file for the film.

However it is necessary for the data being generated by the `CaptureCanvas3D` to be put into a form which a `JMF Processor` can accept. This is achieved by the `ImageSourceStream` class. The `ImageSourceStream` implements the `PullBufferStream` interface which means that it provides data in frame sized blocks, each block being a `JMF Buffer`. Internally the stream maintains an ordered collection of `Buffers` which correspond to the film being captured. This collection is accessed by two methods:

- `void addImage(Image image, boolean lastImage)`
- `void read(Buffer buffer)`

A call to `addImage` results in `image` being converted into a `Buffer` and appended to the collection. The `lastImage` flag is necessary to signal to the stream that no more `Images` will be added. This causes a stream terminating `Buffer` to be sent once all of the `Buffers` in the collection have been read.

A call to `read` results in the data from the next `Buffer` in the collection being copied into `buffer`. In order to ensure that the films generated play back at the correct speed, a time stamp is calculated based upon the frame rate and the number of frames so far. This time stamp is applied to `buffer` to fix its position within the film.

If the collection is empty when a call to `read` is received then one of two things can happen:

- If the `lastImage` flag has been received then `buffer` is returned empty, with an `EOM` flag to signal that the stream is finished.

---

67 For full details consult the JMF API specification of `javax.media.Processor`.
68 For full details consult the JMF API specification of `javax.media.protocol.PullBufferStream`.
69 For Full details consult the JMF API specification of `javax.media.Buffer`.
70 Without time stamps, playback speeds varied greatly despite a frame rate being specified.
71 End Of Media.
• Otherwise the reading thread blocks until the collection is repopulated by a newly rendered and captured frame.

The ImageDataSource class is a PullBufferDataSource\textsuperscript{72} which contains an ImageSourceStream as its only source stream. As a data source it encapsulates this ImageSourceStream in a format ready for JMF processing. This means that the ImageDataSource can be used by any Processor\textsuperscript{73} to view or transcode captured data from the FilmEditor3D.

10.3 FilmGenerator

The FilmGenerator class is a customised Thread\textsuperscript{74} that actually performs the work of encoding the JMF Buffers obtained from the ImageDataSource into a film. The FilmGenerator is instantiated for a particular ImageDataSource and destination file and undergoes the following initialisation stages to prepare for processing the source:

1. A Processor is created to transcode the data and is transitioned into its configured state so that processing options may be set.

2. The content type for the Processor is determined by the format of the output file. Currently only FileTypeDescriptor.MSVIDEO and FileTypeDescriptor.QUICKTIME are used.

3. The formats which the Processor can generate from the input format are checked to find one which is compatible with the specified content type.\textsuperscript{75} Provided a suitable format is found, it is selected.

4. The options are set, so the Processor is transitioned into its realized state and a DataSink\textsuperscript{76} is created for the specified output file.

The Processor, once started, runs in a thread of its own. However the thread that starts the processor must wait until processing is finished in order to close the DataSink and free up other resources. To prevent the event dispatching thread or any rendering threads from being delayed in this manner the Processor needs to be started by a separate, dedicated thread.

By having the FilmGenerator as a thread, the event dispatching thread can start the FilmGenerator and carry on processing. The new FilmGenerator thread can then start the Processor and wait for it to finish. Once processing is completed the FilmGenerator thread performs the necessary closing operations and terminates. The status of the different threads throughout the Processor’s lifetime are shown in Figure 7. Note that the event dispatching thread is able to continue execution even once the Processor has been started.

10.4 CaptureCanvas3D

The CaptureCanvas3D class extends the Canvas3D\textsuperscript{77} class by adding capture functionality. This class is at the core of the fe3d.capture package and brings all of the elements

\textsuperscript{72}For details consult the JMF API specification of javax.media.protocol.PullBufferDataSource.

\textsuperscript{73}For details consult the JMF API specification of javax.media.Processor.

\textsuperscript{74}For details consult the Java2 API specification of java.lang.Thread.

\textsuperscript{75}The success of this step is dependent on the range of codecs and plug-ins available to JMF.

\textsuperscript{76}For details consult the JMF API specification of javax.media.DataSink.

\textsuperscript{77}For details consult the Java3D API specification of javax.media.j3d.Canvas3D.
described above together to form an effective capture system. The capture functionality of the canvas is invoked using the three methods shown:

- `beginLiveCapture(File dest, float frameRate, float quality, int frameCount)`
- `endLiveCapture()`
- `captureStill(File dest, float quality)`

A call to `beginLiveCapture` results in the following steps:

1. A new `Raster` object is created with dimensions, colour model etc. set to match those used by the canvas.

2. The file type of `dest` is checked to determine what format the output should take.

3. If the output is to be a sequence of JPEG images then:
   (a) The quality of the JPEG encoding is set to be `quality`.
   (b) The `stillCapture` and `liveCaptureAsJPEG` flags are set.

4. If the output is to be a film then:
   (a) A new video format is defined with dimensions and colour model to match those used by the `Raster` object and with `frameRate` as the frame rate.
   (b) A new `ImageDataSource` is created to stream the captured images as they become available.
   (c) A new `FilmGenerator` is created for the `ImageDataSource` and destination file `dest`.

![Figure 7: Thread status of FilmGenerator](image)
(d) The FilmGenerator is started and the liveCapture flag is set.

A call to endLiveCapture results in the appropriate capture flags being reset. A call to captureStill is treated much the same as live capture of JPEG images except that the liveCaptureAsJPEG flag is not set. This means that a frame will only be captured once rather than repeatedly. In this instance dest and quality are again the destination file and quality of encoding respectively.

The actual image capture occurs in the Canvas3D postSwap() method which is overridden by the CaptureCanvas3D to include capture functionality. This method is invoked by the renderer every time it completes the rendering of a frame. Placing the capture code in this method allows the retrieval of fully rendered images, whereas code located elsewhere could just as easily capture a partially rendered, or even blank, image.

The capturing itself is relatively straightforward. The Raster object created when capture began is used to retrieve an Image from the frame buffer. Then depending on which flags have been set the Image is either encoded as a JPEG or added to the ImageDataSource as required. When a sequence of JPEG images is created the individual images are numbered sequentially to aid in reconstruction of the play sequence.

10.5 Capture Actions

There are two AbstractActions in the fe3d.capture.action package that are associated with the capturing facilities of the CaptureCanvas3D. These are the LiveCaptureAction and the CaptureStillAction. They both operate by querying the CaptureDialog for the relevant settings and then calling the capture methods described in the previous section with the retrieved settings as parameters.

The LiveCaptureAction also makes use of the fe3d.sequence.Player by initiating playback once live capture has begun. Live capture is ended automatically when either the user stops the Player, or the end of the current play sequence is reached.

11 Testing

The FilmEditor3D is very much an event driven program and as such testing has been performed interactively by ensuring that each individual component of the system performs its job correctly:

- The fe3d.io package has successfully loaded every True 3D Film which has been made available to me. The input and output of FilmEditor3D films has also been checked extensively and no errors have been found.

- The fe3d.background package has a limited scope for error but has been tested with a variety of different background models. The background dialog has also been tested with every possible combination of key presses.

- The fe3d.sequence package also behaves as expected for all of the models and film sequences available to me.

- The fe3d.capture package has one bug that I have been unable to eliminate. When capturing live data the first frame is always captured twice. I believe this to be as a result of the canvas redrawing itself in response to the cursor changing, thus generating an extra image before playback has begun.
In addition to this, informal acceptance testing by my project supervisor at project meetings suggests the project is quite robust and reliable, although the high system requirements can starve other running processes. There is also an unexplained flickering of the GUI on some machines but it is almost impossible to isolate because of two machines running the same operating system, with the same java APIs only one exhibits any unusual behaviour.

12 Evaluation

The FilmEditor3D has met all of the requirements specified for a platform independent tool capable of visualizing True 3D Films. The only observed limitations are high system requirements, and an inability to force exact timing during playback. There are one or two aspects that could benefit from further work:

- Texture loading still generates too much unnecessary data.
- Capturing films slows down the renderer quite significantly and reduces the level of interaction.
- Absolute frame rates can not be guaranteed.

But other than these the FilmEditor3D is a useful and fun visualization tool for working with 3D films.

13 Acknowledgements

I would like to thank the Java3D online community\(^78\) who were very helpful through the initial stages of the project when Java3D was a relatively foreign environment. Particular thanks go to Kelvin Chung and Doug Twilleager from the Sun Java3D team who made bug fixes and beta releases of code available to me in a very timely manner. Thanks also to Fred Klingener for advice on timing issues\(^79\).

\(^78\)http://www.j3d.org and JAVA3D-INTEREST@JAVA.SUN.COM
\(^79\)Fred’s analysis of timing scenarios with Java3D can be found at http://www.brockeng.com/VMech/Time/PlanJ/Clocks.htm.