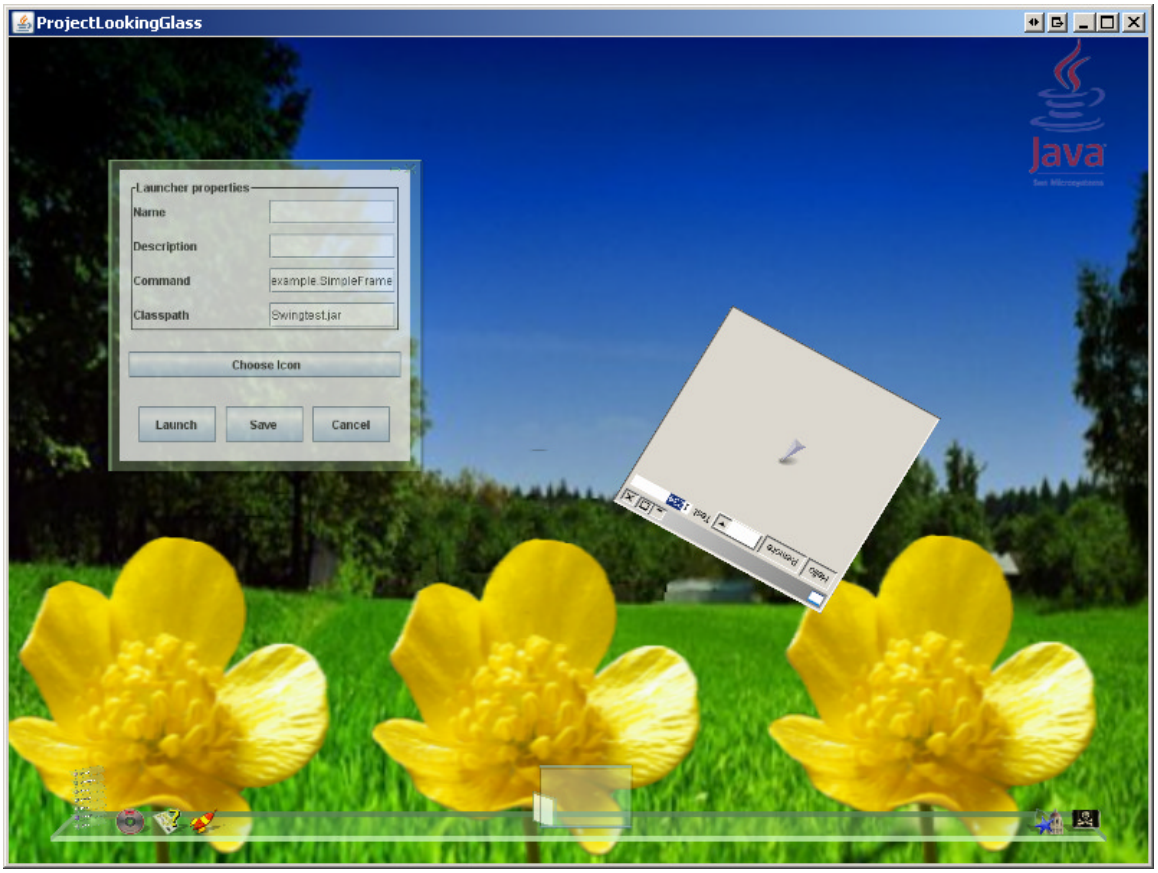
Diploma Thesis

Feasibility Study and Prototypical Implementation of a Window Framework for Digital Tables



Author: **Sascha Hoemig**

Student at the

University of applied Science

Mannheim

Practical implementation at the

University of Calgary

Table of contents

Abstract

Digital Tables have horizontal oriented Monitors, usually many LCD displays connected to display a high resolution desktop. In the future, these tables will be used as a utility to improve meetings by providing a personal computer that is reachable for every team member at the same time. As the table is run from every side at the same time, there are some additional features needed to ensure a smooth workflow. Features that are missing at a normal personal computer are e.g. the concurrent input using multiple keyboards and mice. The other lack that comes up at a digital table is topic of this thesis. All windows must be readable and easily operated from every side of the table. Because of this, the rotation of windows is needed for a digital table. This thesis shows the problems and solutions that come up while implementing a prototype application that can be rotated.

The goal of this thesis is to implement the ability of rotating an Standard Widget Toolkit (SWT) window in one of the following environments: Microsoft Windows, Linux, Mac OS or a (platform-independent) virtual desktop environment. Implementation language is Java. Everything that cannot be done in Java (e.g. Operating System calls) is connected using the Java native interface.

Motivation

This thesis will show the basic ideas of implementing rotatable SWT windows. Every Operating System has its individual problems that make it hard to implement the rotation of a window. The rotation of a window is an essential feature for digital table, thus this problem has to be solved in some way. As the digital table provided by Smart technologies is based on eight LCD displays, each having a maximum resolution of 1280 by 1024 pixel, the hardware issues of performance cannot be excluded while solving this problem. The best solution would be to change the Windows code, at least for the class CWindow, which handles the graphical user interface (GUI) properties. Windows is the most suitable Operating System for the digital table, as currently only Windows drivers are available for the special input provided by the digital table. This thesis is limited to the rotation of Java SWT windows, because SWT applications are currently state of the art as the eclipse development environment introduced the Operating System oriented GUI.

As the SWT framework is very close to the Operating System in its implementation, after the rotation of a SWT window it is only a small step to the rotation of native windows.

Fundamentals

Digital Tables

Picture 1 - A digital table

A digital table is a table with multiple displays that are horizontally oriented and connected in a way to present an enlarged desktop. The main purpose of digital tables is to provide a planning tool for multiple persons at the same time. To make the input easier most tables provide a simulation of a touch screen overall display area.

Technical Specification

The digital table is build out of eight LCD displays, each having a maximum display resolution of 1280 by 1024 pixel. The overall resolution sums up to 5120 by 2048 pixel. To support 3D acceleration in this high resolution the system has two Matrox QID Pro quad graphics cards each connected to four displays. The Windows desktop is extended to all eight displays; this makes it possible to run applications in full screen mode stretched over all eight displays.

To interact with the Operating System in this case Windows XP, the table has on the one hand the normal Mouse and Keyboard and on the other hand, an optical input that simulates a touch screen. To simulate a touch screen on all eight LCD monitors the table has a camera in every corner and additional infrared sensors on the long side of the table. This input device can be calibrated via software to support the different desktop resolutions.

Features

A special feature of the digital table is the stated full screen OpenGL support. In reality there seems to be a lack of display memory to support full screen OpenGL as the OpenGL area is cut off after about three and a quarter displays in length (after four times 1024 pixels).

Two different modes are supported: Firstly, the normal mode that makes the table usable like every other personal computer: one mouse and one keyboard are used as input devices. The second mode supports writing and marking with virtual pens. The table provides four virtual pens and two virtual rubbers. As soon as one of these input devices is taken out of its rack, the table switches to the second input mode. In this mode two pens can be used at the same time, the cameras and infrared sensors are used to differ between the two pens, but as soon as those two pens come too close to each other the cameras can't differentiate between them and some times the identifiers are switched.

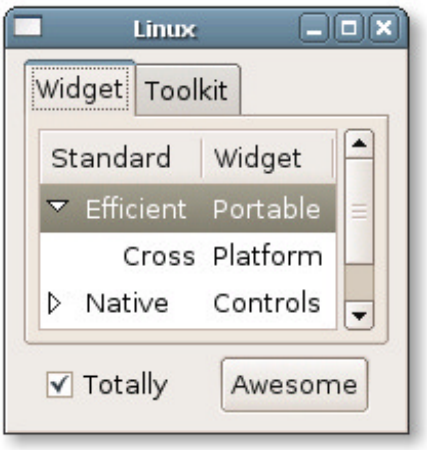
Java Eclipse SWT

The JAVA Eclipse Standard Widget Toolkit (SWT) is a Java Graphical User Interface (GUI) Toolkit that uses the Operating System specific widgets to create a GUI interface. In the past creating a GUI with Java gave the GUI its individual Java style (e.g. using AWT or Swing). Using SWT lets the GUI look like every other Windows (or Linux/MacOS) program. But not only the look and feel of the applications is improved using the SWT framework; as SWT simply wrappes all the Operating System calls the performance and stability is improved compared to a Swing application. The SWT toolkit comes for three different Operating Systems: Windows (Picture 2), Linux (Picture 4) and Mac OS (Picture 3).

Picture 2

Picture 3

Picture 4



The look and feel for a Java application is the same as for non-Java applications.

Because of this, a Java application looks different on every Operating System. The advantage of having fast and platform independent code brings the disadvantage that the same widgets might use different sizes in different Operating Systems.

Looking at the pictures there is a visible difference: The Mac OS X Widget is the only one that needs a vertical scrollbar. All of those little differences make it harder to program a platform independent GUI with SWT.

Each version has its own C code wrapping the system calls. It is connected using the Java native interface, which will be discussed later. The same Java code is shown to the end-user to make the application code remain platform independent as long as an SWT toolkit is available for the different Operating Systems.
Large scopes of applications for SWT are Eclipse Plug-ins. As every Eclipse plug-in is based on the Eclipse Framework, this is based on SWT and uses the SWT toolkit to create its GUI.

Java native Interface

"The **Java Native Interface** (**JNI**) is a programming framework that allows Java code running in the Java virtual machine (VM) to call and be called by native applications (programs specific to a hardware and Operating System platform) and libraries written in other languages, such as C, C++ and assembly.

The JNI is used to write native methods to handle situations when an application cannot be written entirely in the Java programming language such as when the standard Java class library does not support the platform-specific features or program library. It is also used to modify an existing application, written in another programming language, to be accessible to Java applications. Many of the standard library classes depend on the JNI to provide functionality to the developer and the user, e.g. I/O file reading and sound capabilities. Including performance- and platform-sensitive API implementations in the standard library allows all Java applications to access this functionality in a safe and platform-independent manner. Before resorting to using the JNI developers should make sure, the functionality is not already provided in the standard libraries. "[1]

As some types in Java are different to the types in the native language, there are methods that interchange the types into each other. Examples are Strings or Arrays:

```
//Get the native string from javaString
const char *nativeString = env->GetStringUTFChars(env,javaString, 0);
```

Some types stay the same, these types are:

Native Type	Java Language Type	Description
bool	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits

1 From Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Java_Native_Interface

long	jint	signed 32 bits
long long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits

All these types can be used without an explicit cast.

The Java Native Interface will be used whenever Java does not provide the needed functionality. This will be the case as soon as Operating System calls are needed to interact with a window. [1]

Java3D

In the change from version 1.3.1 to 1.3.2 Java3D became a community source project developed on java.net. In the past Java3D had no power to compete, however the current stable version 1.5.0 is much faster than the old version. It is now completely based on OpenGL, which was introduced in the year 1992 and is one of the most widely used 2D and 3D graphics application-programming interface (API). With the help of OpenGL's hardware-accelerated rendering pipeline Looking Glass is able to transform the 3D scene graph objects even at the very high resolution of the digital table.

Java3D is now divided into several sub-projects. Since in the past everything was developed under one name, SUN kept a parent project that has several related child projects. It is available at https://java3d.dev.java.net and links to all subprojects. The main classes are being developed under the name *j3d-core* available at https://j3d-core.dev.java.net.

SUN kept the existence of the mathematical API for 3D programming, but made it a child project of java3D. It is available at https://vecmath.dev.java.net. Other maintained projects in relation to java3D are:

- j3d-contrib-utils Contrib (optional) utilities for Java 3D
- j3d-core The Java 3D API Core
- j3d-core-utils Core Utils for Java3D
- j3d-examples Java 3D Example Programs

- j3d-incubator Java 3D Incubator Project
- j3d-vrml97 Java 3D VRML97 Loader
- j3d-webstart Java 3D Web Start Binaries
- j3dfly J3DFly and J3dEditor sample applications
- skinandbones Real time mesh deformation using skeletons.
- vecmath The vecmath package
- vecmath-test JUnit tests for the vecmath package

A Java3D scene is based on the underlying scene graph. "The scene-graph is an object-oriented structure that arranges the logical and often (but not necessarily) spatial representation of a graphical scene. The definition of a scene-graph is fuzzy, due to the fact that programmers who implement scene-graphs in applications and in particular the games industry take the basic principles and adapt these to suit a particular application. This means there is no hard and fast rule as to what a scene-graph should or should not be.
Scene-graphs are a collection of nodes in a graph or tree structure. This means that a node may have many children but often only a single parent, the effect of a parent is apparent to all its child nodes - An operation applied to a group automatically propagates its effect to all of its members. In many programs, associating a geometrical transformation matrix (see also transformation and matrix) at each group level and concatenating such matrices together is an efficient and natural way to process such operations. A common feature, for instance, is the ability to group related shapes/objects into a compound object which can then be moved, transformed, selected, etc. as easily as a single object."[1] [6]

1 From Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Scenegraph

Prearrangements

Operating Systems

As the goal of this thesis is the implementation of a Framework that supports the rotation of windows, the first step is the look into the common Operating Systems, if they already support the rotation of windows or which Operating System is the best one to change to support this feature.

WindowsXP

The look into WindowsXP pretty fast excluded this Operating System to support the needed feature. WindowsXP does only support rectangular bounding boxes that are axis parallel. As rotated windows need rotated bounding boxes to support dipping, the implementation of a bounding polygon that does not need to be axis parallel is needed for WindowsXP.
These changes could be made by subclassing the CWindow class of Microsoft Windows, but as the WindowsXP source code is not available to public, using WindowsXP as the preferred Operating System seems to cause more trouble than an open source Operating System. [2]

Linux

"If it can't be done with Linux, it can't be done."[1] – Sounds good, so let's take a closer look into Linux. The SWT framework for Linux is based on GTK. GTK has the same problem as the window framework of WindowsXP: It does not support bounding boxes that are not axis parallel. As Linux is open source the look into GTK shows, that it is based on GDK which is based on the X server framework. The needed changes to support window rotation should then be implemented to the X-Server framework and made available to GDK, then GTK and in the end to SWT. The substantial features that need to be added to the X-Window system would at least be:

- Bounding polygons

[1] Signature of Sid Boyce at Linux Today (http://www.linuxtoday.com/)

The rotated window will have a rectangular non axis-parallel bounding box. As the clipping is done with the axis-parallel bounding box by a simple “x, y within range check” the implementation of Bounding polygons would need the implementation of:

- Clipping with polygons

  The clipping with polygons, as implemented in every 3D environment, is needed whenever a rotated window is placed in front of another window. It decides whether a pixel of the background is visible or not.

- Local Coordinate System

  As a mouse click to the x,y-coordinates of the window needs to be transformed to the rotated x,y-coordinates, every window needs a transformation matrix that stores the current rotation.

As the complete Linux source code is implemented in C and my C knowledge is not as good as needed, I looked into the next Operating System. [3]

Mac OS

As Mac OS X is the first 3D desktop environment to look into, it is assumable that rotating windows should be easier than in a 2D Operating System. Mac OS X is based on OpenGL and has the ability to animate the windows when they are closed or minimized. After taking a deeper look into Mac OS X the Operating System appears to be a normal 2D Operating System with additional visibility features that add some 3D style to the windows. The real OpenGL API is hidden to the user and can’t be used that easily. All animations shown by the Operating System are hard coded animations that only work on a picture. No real window (with the ability of handling input) is transformed, instead a snapshot of the application is made and transformed whenever the window is minimized or closed. For that reason Mac OS X disqualifies to be a suitable Operating System to get extended to a window framework for digital tables. [4]

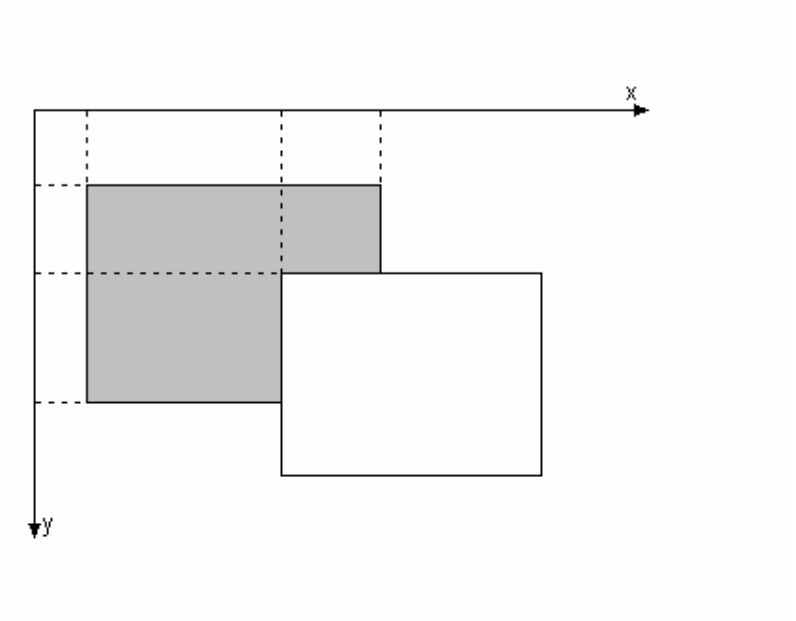
Outcome of the Operating System selection

By default, none of these Operating Systems has the ability to rotate windows.

Indeed it is possible to implement the rotation itself, as this is nothing else than the rotation of a picture. As the mouse can enter or leave the rotated area (window) at any side and place, the coordinate system transformation must be applied to the mouse cursor position whenever the mouse enters the window.
These transformed coordinates can then be calculated back to the real window coordinates, which are needed to send the correct events.

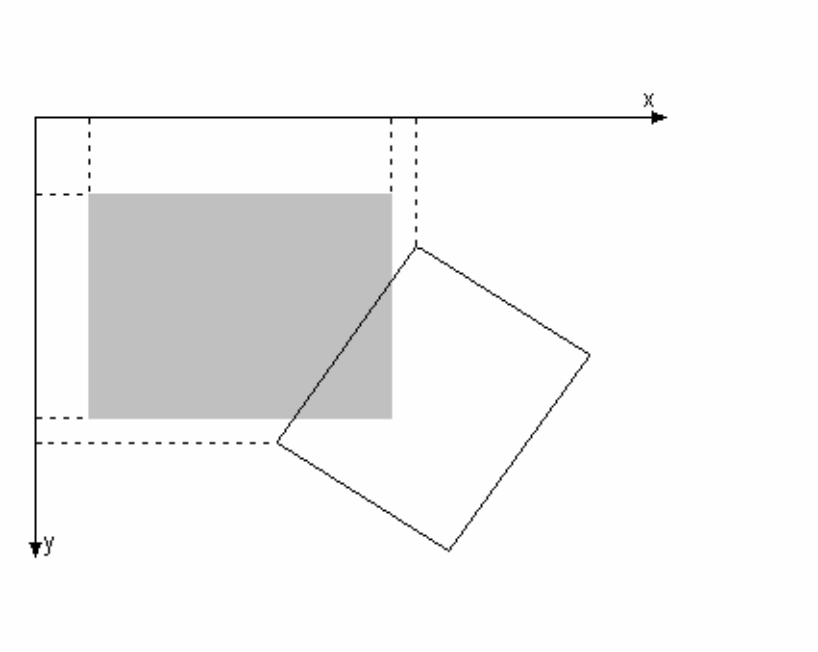
These two things (the rotation and the mouse input) are extensions on the window framework of MS Windows or Mac OS X, as non-rotated windows are still working normal as before.

The required feature that makes it impossible to implement the rotation directly in the Operating System is the clipping of two windows.
Microsoft Windows' and Mac OS X's clipping algorithm is based on windows that have axis parallel edges. The clipping is done using position, height and width of the window.



Picture 5 - The clipping of two unrotated windows

The Operating System does a quick check if one of the corners of a window is located inside of another window, if so clipping is done. This algorithm does not work for rotated windows.



Picture 6 - No clipping, as no corner is inside the gray window.

This problem cannot be solved within Windows or Mac OS X because the Operating System has to be changed at positions that are hidden because it is not Open Source.

To implement the feature, Linux is the best option as all its source code is available. Another option might be Windows Vista, but at the time I caught up on the Operating Systems, Microsoft had no useful API for Windows Vista available.

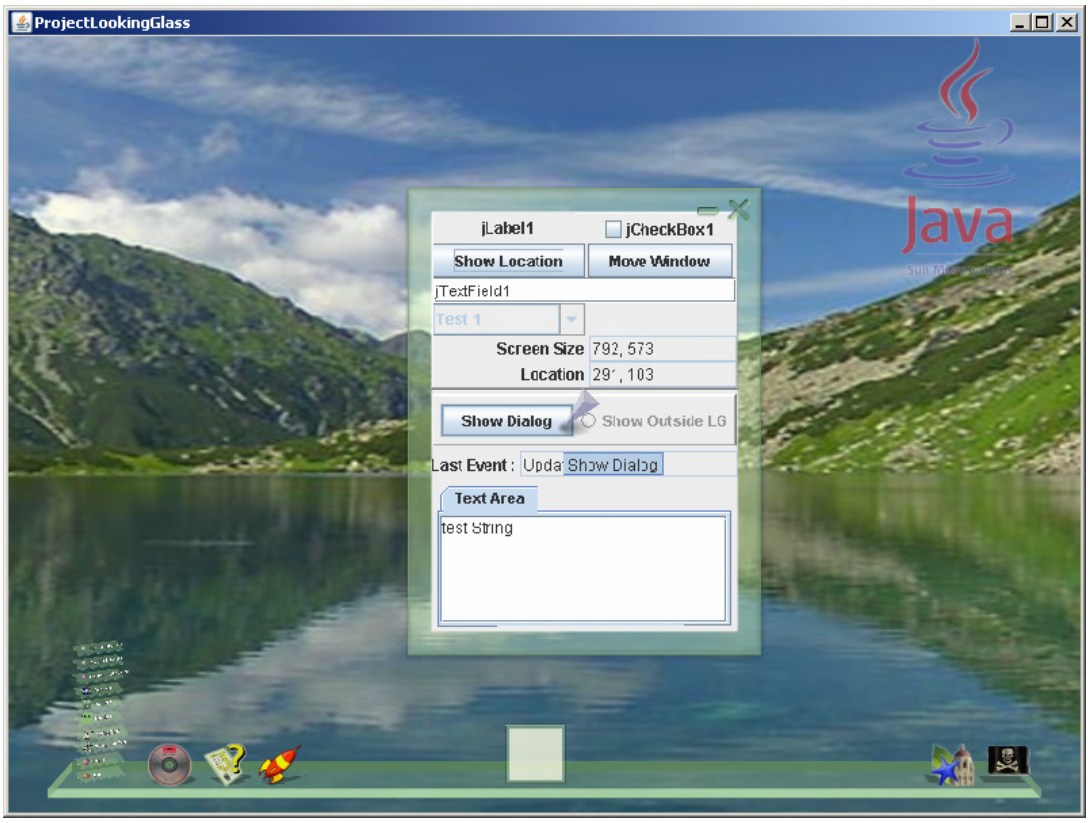
Linux is the best choice if the framework must be implemented directly within an operating system. Not because it is very easy to implement these features in Linux, but rather it is the only operating system (out of Windows, Mac OS X and Linux) where it is possible to change the operating system code.
If there were drivers for Linux that support the additional input features of the digital table and enough time I would recommend to use Linux and change the X-Window-Server's code to support rotation of 2D windows.

Looking at the problem in a realistic way, taking into account that time and manpower is limited the best solution is an extension of an existing 3D framework, where Looking Glass is a very good choice as it is based on Java.
Additionally, as there are only WindowsXP/2000 drivers available that support the input using pens or fingers at the digital table, it would be good to have a framework that sits on top of Windows. Otherwise, drivers for the Operating Systems Mac OS or Linux would be necessary additionally to the changes within the 3D environment.

The SUN Java Looking Glass Framework

SUN's current project Looking Glass is an OpenGL based window framework that sits on top of the Operating System. It provides an Open Source based 3D desktop environment that is still under development. It is currently under development for the Operating Systems:

- Linux x86
- Solaris x86
- Windows



Picture 7 – SUN's Java Looking Glass Environment showing a Swing application

The framework is completely implemented in Java using SUN's actual version of Java3D. Most features of Java Swing applications can be shown with this framework. Because this framework is platform independent, I choose this framework to make it support SWT applications, which can then be transformed in any way. [7]

Advantages and Disadvantages

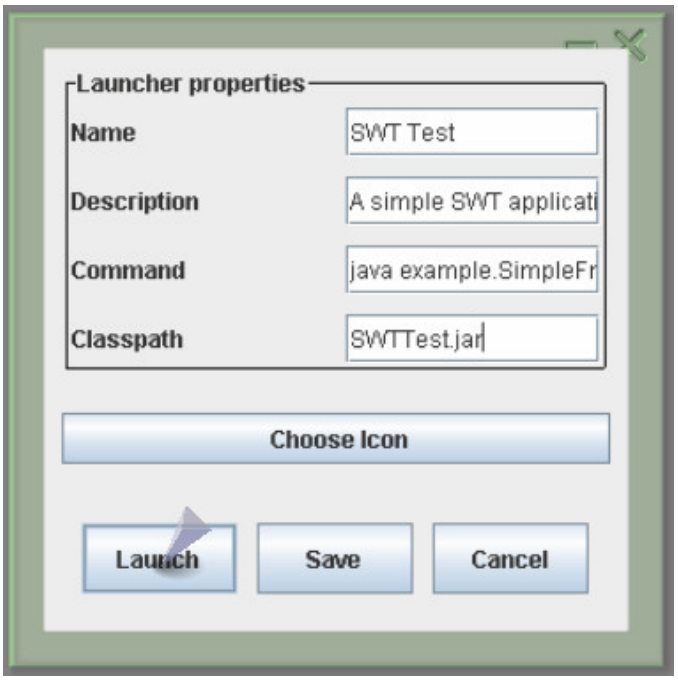
SUNs Looking Glass is completely implemented in Java. Because of this, the 3D framework is completely platform independent, as long as all OpenGL features are supported.
As the whole framework is open source, it is very easy to implement changes that might be necessary in the future

Concerning the software side, there are no disadvantages. Looking Glass is well structured and provides good documentation and support through the large community.
The Looking Glass implementation uses Java3D to create a 3D desktop environment providing a task bar and a start menu.

The probably biggest disadvantage of SUN's Looking Glass are the high requirements on the hardware. On the other hand, as many Operating Systems are moving towards a 3D accelerated user interface, in the future the hardware will adapt and support Looking Glass features even at the high resolutions a digital table can provide. The Matrox QID Pro graphics card, that is currently used for the table is a very good 2D graphics card, but cannot compete in 3D acceleration to the card available from ATI or NVIDIA. The problem with the cards available from ATI or NVIDIA is that most cards do only support two monitors. The current system would then need four graphic cards. Up to now I am not aquatinted of a mainboard providing four PCIe slots though.

Implementation

The Looking Glass Framework already provides an application called "Launcher" that executes java applications out of the framework.



Picture 8 – The "Launcher" application

As Looking Glass can run all applications inside the framework in the Linux version, but there are now Linux drivers for the digital table, I concentrate on the Windows version that still lacked this feature and cannot execute other applications than Java Swing applications or applications written especially for Looking Glass.
Looking into the code that creates the 3D shape for the Swing applications did not help very much, as Swing applications are completely independent from the underlying Operating System.

The basic idea of creating a 3D representative of a 2D SWT window is to get the image of the 2D application and use it as a texture to be mapped onto the 3D shape of appropriate size. This texture should then be replaced whenever the image of the 2D application changes. If this is done the direction 2D $\Rightarrow$ 3D is implemented completely.

The direction 3D ⇒ 2D is more effort. All keyboard and mouse events must be send to the right widget and all system events (e.g. resizing the window) must be handled correctly.

2D image capturing

Implementation Try-Outs

An easy way to capture the image of the 2D application is to create a screen shot of the application whenever the Operating System does a repaint on the window. Using this method, only works if the 2D application is fully visible all that time during the repaint. Trying to capture the image of a window that is (partly) hidden results in erroneous image of the correct size, but with the current view of the screen meaning containing the window that is on top of the (partly) hidden one.
Moving the window out of the visible screen area, leaving it there uncovered does not result in a correct image neither, because Windows discards all WM_PAINT[1] messages to that window as soon as it is not visible any more.

"The WM_PAINT message is sent when the system or another application makes a request to paint a portion of an application's window. The message is sent when the UpdateWindow or RedrawWindow function is called, or by the DispatchMessage function when the application obtains a WM_PAINT message by using the GetMessage or PeekMessage function."[2]

Working Implementation

As all the other tries, this implementation is again done with the Java native interface. Implementation language is C.
The Windows API provides a window event called WM_PRINT. "The WM_PRINT message is sent to a window to request that it draw itself in the

[1]

[2] MSDN Library http://msdn.microsoft.com in "Win32 and COM Development → Graphics and Multimedia → GDI → Windows GDI → Painting and Drawing → Painting and Drawing Reference → Painting and Drawing Messages

specified device context, most commonly in a printer device context."[1] In this case, the WM_PRINT command is not used to draw the window content into a printer device context. When the application is started, a memory device context is created that behaves like the device context, into which Windows is able to draw using the WM_PRINT message.
Now I am able to use the WM_PRINT message to fake Windows and make it draw the content of the window into a memory device context.

Java SWT changes

Now that we have a working image capturing algorithm, the next step is to change SWT to support that feature.
As this is a prototype implementation only the windows representing a Shell are visible within the Looking Glass 3D framework. All other windows e.g. dialog boxes or popup menus are currently not transformed into the 3D framework.
To register every Shell with the corresponding 3D implementation the SWT code is changed at the constructor of the class *org.eclipse.swt.widgets.Control* that is a super class of *org.eclipse.swt.widgets.Shell*.

The attribute that represents the 3D window is added to the class:

```
import org.jdesktop.lg3d.displayserver.swt.SWTWindow3D;
[…]
public SWTWindow3D swtWindow3D = null;
```

This attribute is initialized as soon as one of the constructors of the class Shell is called.

```
swtWindow3D = new SWTWindow3D(display, this);
```

No other changes were made to the SWT code. Everything else that is necessary, e.g. the interaction with the 2D window, is done through Java native

[1] MSDN Library http://msdn.microsoft.com in "Win32 and COM Development → Graphics and Multimedia → GDI → Windows GDI → Painting and Drawing → Painting and Drawing Reference → Painting and Drawing Messages

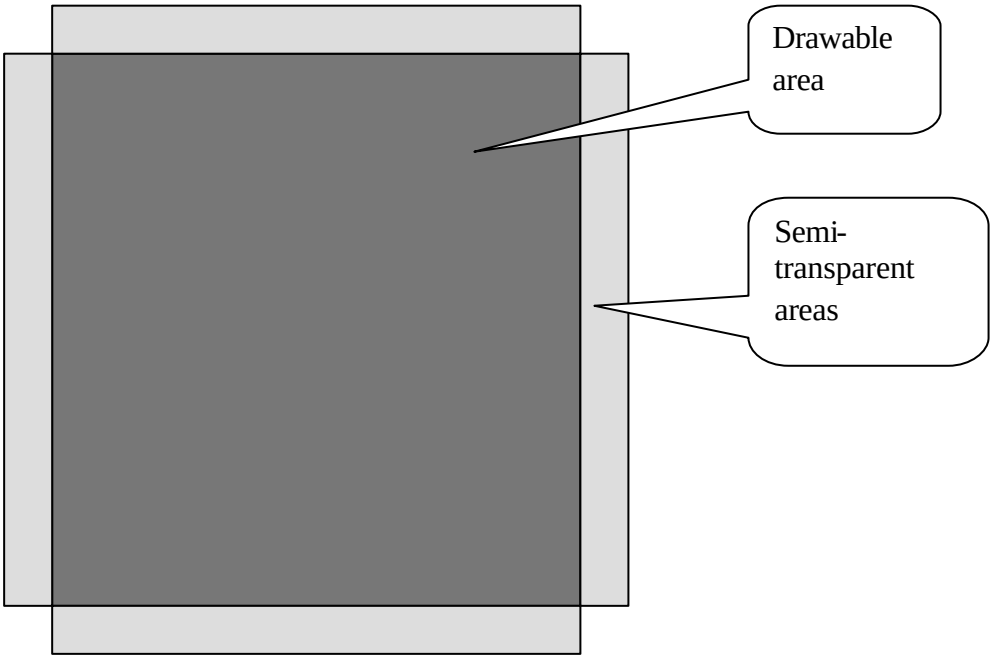
code, because the direct connect to the Operating System is much faster then running through the SWT code, which in the end needs the system call anyway.

SUN Java Looking Glass changes

As the SUN Looking Glass environment is still under development, which means there can be code changes every day, no changes to the Looking Glass code itself where made. All needed implementations where done to additional classes located in the package *org.jdesktop.lg3d.displayserver.swt*.
The main class in this package is the class that represents the 3D Shell, it is called *SWTWindow3D.java.*

SWTWindow3D.java

The SWTWindow3D class extends the already provided class Frame3D. This class represents the 3D twin of the 2D SWT Shell. The shape of the Window is created using a FuzzyEdgePanel again provided by Looking Glass.



Picture 9 – Draft of a FuzzyEdgePanel

The current implementation sets the height and width of all semi-transparent areas to zero. The decision to use the *FuzzyEdgePanel* instead of a simple rectangular shape was made, because the edges can be used to interact with the window, like moving, scaling, rotating or even more complex transform operations.

The appearance that is added to the shape has nothing else than a texture associated. This texture is the image of the 2D SWT application.

In the OpenGL version which the graphic card supports only textures with a height and width that are a power of two are allowed. Because of this constraint, the image of the SWT application must be transformed (scaled) to the next power of two value. As the 3D application in general has no dimensions of a power of two, the texture is again transformed within the rendering pipeline by the texture-coordinates.
The newest OpenGL specification allows textures of whatever size. The try to use textures that do not have a width and height of a power of two worked, but dropped the performance too a minimum. This looks like the graphic card (or the driver) of the digital table does not support the latest OpenGL features.

The official OpenGL specification says:
"I.3 Non-Power-Of-Two Textures
The restriction of textures to power-of-two dimensions has been relaxed for all texture targets, so that non-power-of-two textures may be specified without generating errors. Non-power-of-two textures was promoted from the GL ARB texture non power of two extension. "[1]

The class SWTWindow3D also handles the events received from the mouse and keyboard and sends them to the SWT application or directly to the Operating System. Because SWT does not allow another thread than the thread that manages the redraw to interact with the GUI, it is needed to spawn a thread

[1] Taken from: "The OpenGLR Graphics System: A Specification", (Version 2.1 - July 30, 2006), Appendix I.3, written by Mark Segal and Kurt Akeley, available at: http://www.opengl.org/documentation/specs/version2.1/glspec21.pdf

that is a child of the repaint thread, whenever an interaction with the SWT window is wanted.
For this reason the class SWTWindow3D contains three different inner classes, which all represent one thread and are only created once at the time the 3D window is created. The inner classes are:

- MouseButtonThread
- MouseMoveThread
- RepaintThread

These threads are started every time the associated event occurs and run to the end as a child thread of the repaint thread.
The MouseButtonThread and the MouseMoveThread are very similar, they are used to find the correct widget at the (x, y)-position. As MouseMoveEvents occur very fast after each other the MouseMoveThread has an optimization, it stores the widget, that received the last MouseMoveEvent and tries this widget first for containing the given (x, y)-values. This reduces the complexity to one comparison if the mouse did not move to another widget. Of course the overall complexity stays at O(n) (n equals the number of widgets in this shell) as in worst case every widget must be tested if it contains the (x, y)-coordinates.

The RepaintThread is needed, because with every repaint it is checked whether the size of the 2D application has changed, and if so it changes the 3D applications size as well. The method that gets the height and width of the 2D application again can only be executed within the repaint thread of SWT otherwise an InvalidThreadAccess exception is thrown by SWT.

The common way to use those threads looks like this:

1. Feed the thread with all needed information. These are mostly the (x,y) coordinates.

```
mouseButtonThread.point = getSWTCoordinates(point);
```

2. Set the thread to not done yet.

```
mouseButtonThread.done = false;
```

3. Start the thread as a display thread's child.

```
display.asyncExec(mouseButtonThread);
```

4. Wait for the thread to end.

```
while(!mouseButtonThread.done) {
	try {
		Thread.sleep(50);
	} catch (InterruptedException e) {}
}
```

5. Use the computed information.

```
return new SWTMouseButtonEvent(shell.handle,
                    mouseButtonThread.point.x,
                    mouseButtonThread.point.y,
                    mouseButtonThread.handle,
                    type, wParam, mouseButton);
```

Event classes

During the implementation of the framework, a couple of different event classes needed to be implemented. The package *org.jdesktop.lg3d.displayserver.swt.events contains the following events*:

- SWTKeyEvent
- SWTMouseEvent (abstract class)
- SWTMouseMoveEvent
- SWTMouseButtonEvent

All event classes contain the needed data and provide "get methods", which return the data in a formatted way to be sent directly to the Operating System.

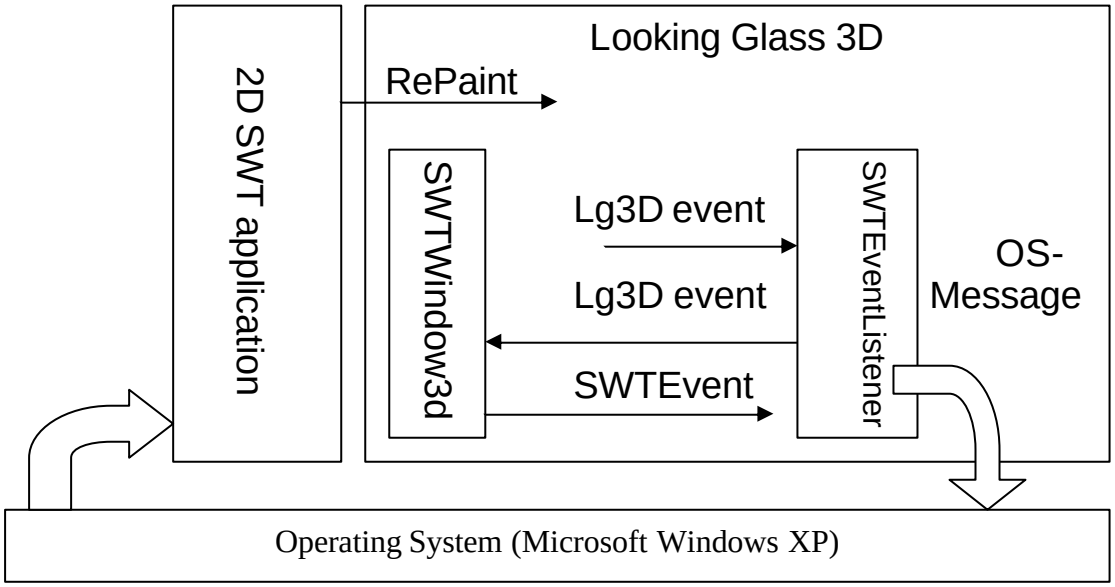
Example:

> Every mouse event needs the (x, y)-values where the event occurred. Microsoft Windows wants the coordinates within one integer value. The upper 16 bit contain the y-value and the lower 16 bit contain the x-value. Both values are stored within an integer field each, but the SWTMouseEvent class contains a get-method that returns one int with the y-value 16 bit shifted left and the x-value added. This simplifies the code as nothing has to be calculated outside of the event classes.

SWTEventListener

The class SWTEventListener implements the interface LgEventListener which is provided by Looking Glass. This class is located in the same package as all the other event classes mentioned before.

All events received from Looking Glass 3D are processed by the listener in the same way:
The event is received from Looking Glass and sent to the SWTWindow3D, which creates an SWT event, that is sent back to the listener.
This SWT event is then sent to the Operating System using the by SWT provided method `OS.SendMessage(...)`.



Picture 10 - Event diagram

The picture above shows the circle of events generated whenever a SWT application is run within the Looking Glass environment. Interaction with the 3D representative SWT window generates an Lg3D event within the framework. The class SWTEventListener, which uses SWTWindow3D to transform the event to a SWTEvent, receives this event and sends the appropriate Operating System Message. This message is received by the 2D SWT application and

might cause a redraw. This repaint event will then cause the 3D environment to obtain the new image of the application.

The SWTEventListener class registers itself at the Looking Glass environment to listen to the following events:

1. **MouseButtonEvent3D**

   Whenever a mouse button is pressed an event is created by the Looking Glass environment. This event contains an attribute type that differentiates if the event is a left single-, left double- or right-click.
   Depending on the event type, the message is created and sent to the Operating System. E.g. the message used to identify a left double click is:

```
OS.SendMessage(handle, OS.WM_LBUTTONDBLCLK,
          event.wParam, event.getCoordinates());
```

   In this case, the handle represents the handle to the widget at the coordinates the double click occurred.
   OS.WM_LBUTTONDBLCLK is a constant defined within the SWT API that represents the left button double click. Other possible constants that represent button click events are defined in the SWT class:

org.eclipse.swt.internal.win32.OS

```
public static final int WM_LBUTTONDBLCLK = 0x203;
public static final int WM_LBUTTONDOWN = 0x201;
public static final int WM_LBUTTONUP = 0x202;
public static final int WM_MBUTTONDBLCLK = 0x209;
public static final int WM_MBUTTONDOWN = 0x207;
public static final int WM_MBUTTONUP = 0x208;
public static final int WM_RBUTTONDBLCLK = 0x206;
public static final int WM_RBUTTONDOWN = 0x204;
public static final int WM_RBUTTONUP = 0x205;
```

Currently implemented events are WM_LBUTTONDBLCLK, WM_LBUTTONDOWN, WM_LBUTTONUP, WM_RBUTTONDOWN and WM_RBUTTONUP. This code can be taken as an example to implement any missing mouse button events. After most mouse button events the SWTEventListener forces a full repaint on the SWT window. This is necessary because Windows does not always do a repaint after a mouse click. An example for the missing repaint is a text selection within a textfield using the left button double click.

2. MouseWheelEvent3D

The MouseWheelEvent3D is sent whenever the mouse wheel is turned and the mouse pointer is above the SWT 3D representative window. This event does not only report that the mouse wheel was turned, it also reports if any modifier keys where pressed while the event was created.

The MouseWheelEvent3D is not sent to the Operating System and because of this never reaches the SWT application.

At the moment the mouse wheel is used to interact with the visible style of the 3D window. If the mouse wheel is turned while the *Control* key is pressed the user zooms into or out of the 3D SWT window.

If the modifier key *Shift* is pressed during mouse wheel rotation the application rotates ten degrees for every MouseWheelEvent3D either clockwise or counter clockwise depending on the mouse wheel direction.

3. MouseDraggedEvent3D

A MouseDraggedEvent3D occurs whenever the mouse is moved and one of the mouse buttons is pressed and held during the movement.

A special implementation detail of the MouseDraggedEvent3D is that the handle of the widget that receives the drag events does not change even if the mouse moves out of the widgets area. Again, the textfield can be taken as an example. If you want to select text within a textfield by

dragging over the text, the text still stays correctly selected if the mouse pointer leaves the textfield.

A Button that is pressed is another example, if the mouse leaves the area of a button with its left button pressed, the button raises again. If the left button is then released, the button does not receive a button clicked event. On the other hand, if the mouse enters the area of the button again, the button gets pressed again and will receive the click event if the mouse button is released now.

Again, after a MouseDraggedEvent3D is processed a repaint of the whole 2D SWT window is forced by the SWTEventListener, as Windows does no repaint itself while selecting text, which can be done using a MouseDraggedEvent3D.

The MouseDraggedEvent3D is sent to the Operating System using the command

```
OS.SendMessage(event.handle, OS.WM_MOUSEMOVE, event.wParam,
                    event.word);
```

In this case the modifier “key” contained within the “wParam” of the method is not a real key, it is a mouse button instead.

4. MouseMovedEvent3D

Another event that was implemented is the MouseMovedEvent3D. This event is needed, as Windows creates tooltips on based on the mouse’s position. As an indication that the position has changed mouse-moved-events are fired and received. E.g. tooltips might appear when hovering on top of a widget. Another issue why mouse move events are needed, is, that some widgets change their graphical user interface style when the mouse is moved across them. For example the move over a button causes Windows to do a repaint, because the button might have changed its appearance.

The implementation details are explained in the chapter “Implemented features”.

5. KeyEvent3D

To send a keyboard event to a SWT application, it is necessary to know the widget, which is destination of the keyboard event.
While sending mouse events, the assumption was made, that the widget that should receive a mouse click is at the position of the mouse pointer. This assumption makes sense, as this defines a mouse click. Looking at the keyboard events, there must be a different way of identifying, which widget should receive the keyboard event.
Now the assumption is made that there can only be one widget selected (activated) at a time. This selected widget (the int pointer to it) is stored within the class SWTWindow3D in the field

```
int selectedComponent
```

At the moment the selected widget can only be changed using the mouse. It is not possible to use the implemented Tab order to jump through the widgets.
The event received from Looking Glass can have three different types, they can be identified using:

- `e.isPressed()`
- `e.isReleased()`
- `e.isTyped()`

where *e* is the `KeyEvent3D` received from Looking Glass.

If a key is pressed, the generated event will have the attribute pressed equal true. If a key is released, the generated event will have the attribute released equal true. Looking at the attribute typed, there are two ways to create an event with that attribute equal true.

1. By pressing and releasing a key, the Looking Glass environment creates events with the attributes:
    1. pressed
    2. typed

3. released

2. By continuously pressing a key the environment will create events with those attributes:
    1. pressed
    2. typed
    3. typed
    4. […]
    5. typed
    6. released (when finally releasing the key)

The time between the generated “typed”-events is system dependent. It is mostly configurable within the computers’ Bios.
Depending on the event type, three different Operating System messages are created.

- `OS.SendMessage(e.getHwnd(), OS.WM_KEYDOWN, e.getWParam(), e.getLParam());`

- `OS.SendMessage(e.getHwnd(), OS.WM_KEYUP, e.getWParam(), e.getLParam());`

- `OS.SendMessage(e.getHwnd(), OS.WM_CHAR, e.getWParam(), e.getLParam());`

6. **MouseEnteredEvent3D**

   The SWTEventListener is registered to listen for MouseEnteredEvent3D events, but they are not used at the moment.

   In the future, this event can be used to show the user where the mouse is and which window was selected, as the user easily looses track of the mouse pointer at the high resolution of the digital table.

Java native interface implementations

Not only Java code was necessary to implement the required features. The Operating System events that are sent to the widgets of the 2D SWT application are using the SWT java native code, which calls C code in behind, but the C code that grabs the image every time the GUI of the 2D application changes was self-implemented using the java native interface.

The java class `NativeScreenGrabber.java` located in the package `org.jdesktop.lg3d.displayserver.swt` contains the java code, which calls the C code connected using the JNI.
The function getting the image data is

```
public static native void getDeviceBitmap(int[] ints, int handle,
                                           int width, int height);
```

The parameters are:

- int[] ints

  This array is filled by the function and returns the image data as a bitmap. The advantage of using the parameter as a return type is that if the size of the image did not change (means the application size did not change), no new array has to be created. This brings a large improvement in performance.

- int handle

  This is the handle to the window which shall be grabbed to an image.

- int width, int height
  These parameters specify the size of the image. This is the same as the size of the application.

The C implementation behind this method declaration creates a dynamic link library (dll), which is loaded at runtime.
The C code is more or less still a prototype implementation. There are many people asking on the internet to get an image of an invisible window, but there is still no good solution available.

My implementation works like this:

1. Create a compatible device context.
2. Create a compatible bitmap.
3. Use the message WM_PRINT to draw the window into the created device context.
4. Get the bitmap out of the device context.
5. Get the int array out of the bitmap.
6. Release the memory of the created objects.

The result of the compiled and linked C code is a dynamic link library called ScreenGrabber.dll. This library is loaded using the java native interface:

```
static {
	System.loadLibrary("ScreenGrabber");
}
```

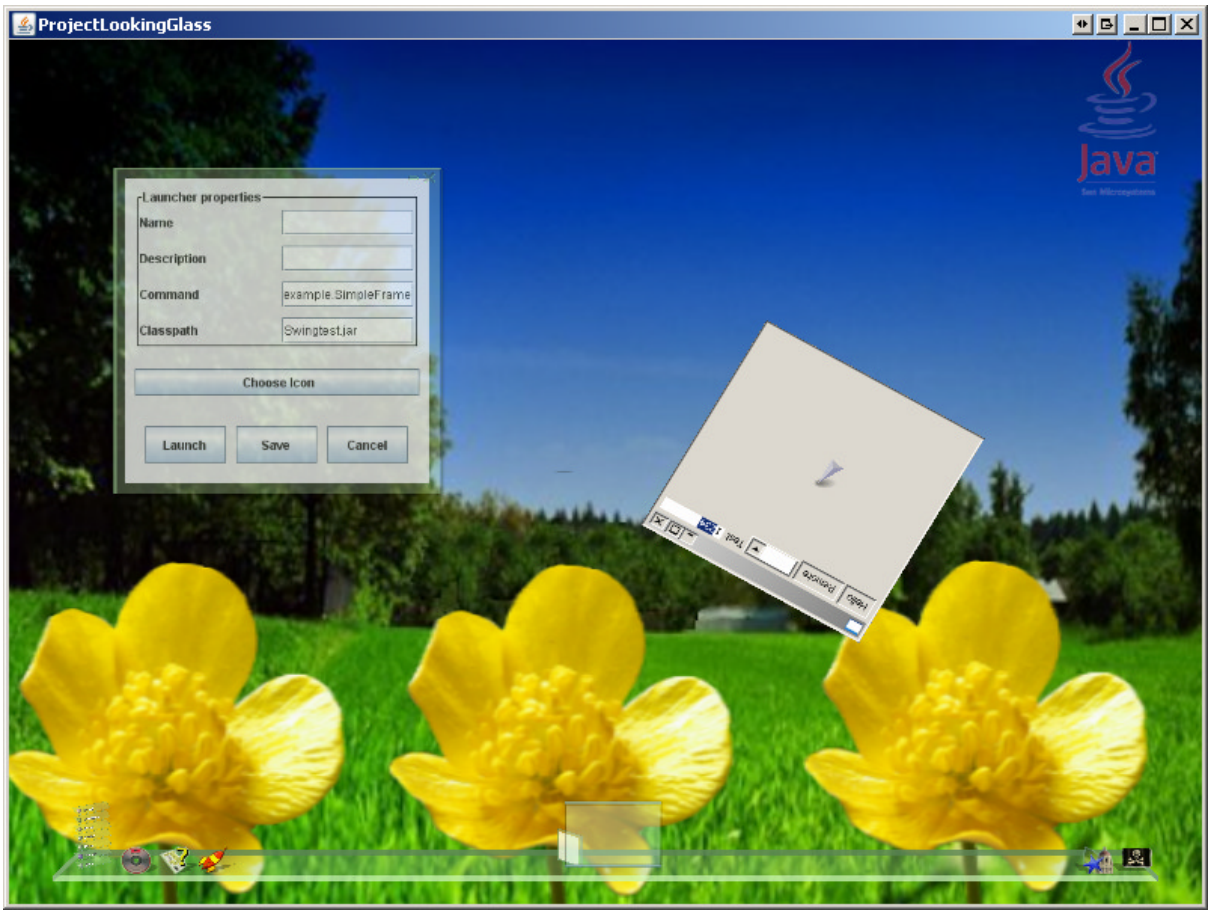
Additionally the C file still contains the first try of capturing an image. The method is called `captureImageToClipboard`. It can be used as a simple example. This method always captures the visible area and all elements (e.g. other windows) above the window to the Windows clipboard. As this method does not work with off screen windows, it is not used any more.

Implemented Features

Rotation

The most important feature is the rotation of the 3D SWT window, as it is topic of this thesis.
The Looking Glass framework provides all the Java3D functions to the software implemented for the framework. As the 3D representation of the SWT window is a scenegraph object, it can be transformed in any way using the Java3D class Transform3D. This class holds a 4x4 matrix that represents a transformation that is added to the partial scene graph below the transformation group (node).

This transformation group can be set to a rotation around the z-axis, which will result in the required rotation.



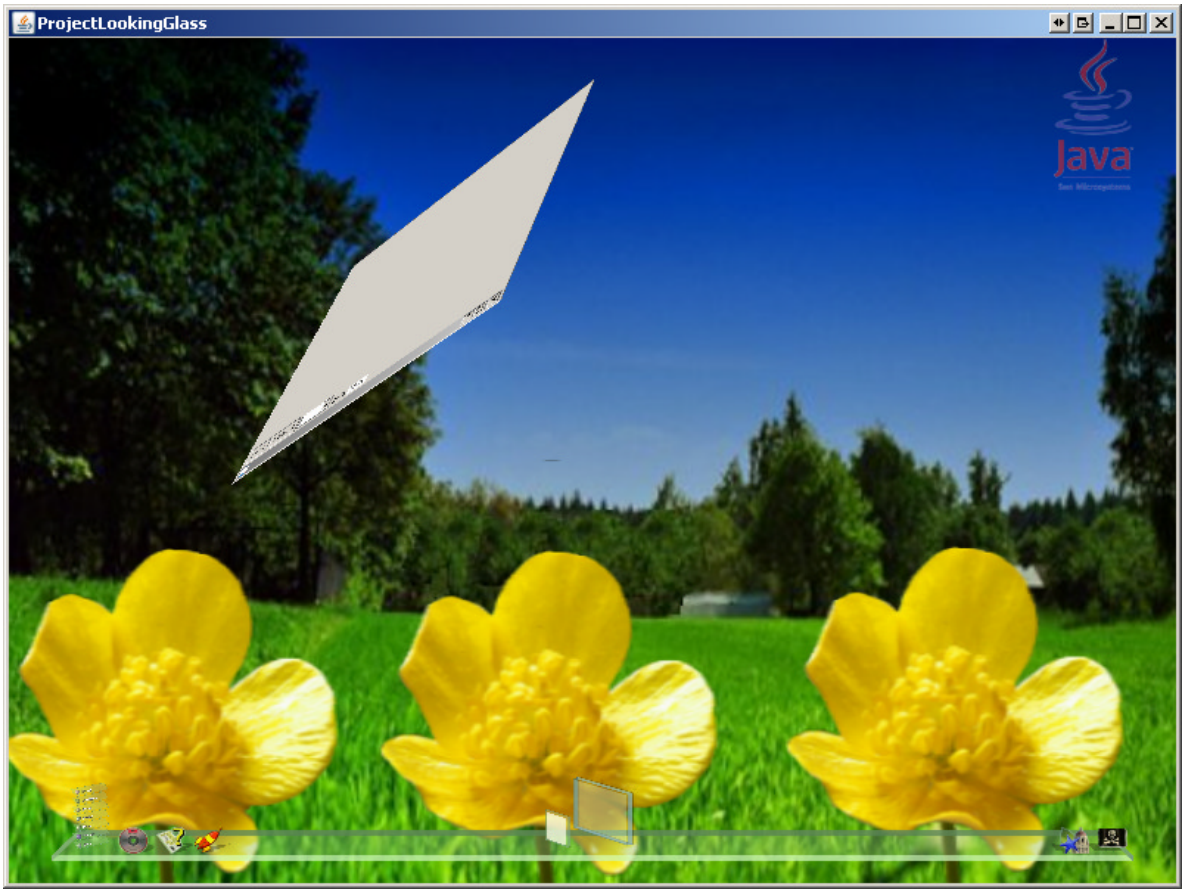
Picture 11 – A rotated SWT window

The picture shows the 3D window rotated around the z-axis by an angle of about 150 degrees counter clockwise.
Every mouse wheel indentation (the atomic event Windows receives from the mouse wheel) rotates the 3D window ten degrees either clockwise or counter-clockwise. To perform this operation without a virtual 3D environment complicated image-rotation algorithms would be needed. Within an OpenGL supporting framework the rotate operation is nothing else than a simple method call.

```
if (mf.length >0 && mf[0].equals(ModifierId.SHIFT)) {
      if (mwe.getWheelRotation()>0){
            lgWindow.getComponent().setRotationAxis(0f, 0f, 1f);
            lgWindow.getComponent().changeRotationAngle(
                        lgWindow.getComponent().getRotationAngle()-
                              (float)Math.toRadians(10d), 200);
      }
      else {
            lgWindow.getComponent().setRotationAxis(0f, 0f, 1f);
            lgWindow.getComponent().changeRotationAngle(
                        lgWindow.getComponent().getRotationAngle()+
                              (float)Math.toRadians(10d), 200);
                        }
                  }
```

The code above reacts on mouse wheel events with the modifier key shift. Whenever the mouse wheel is rotated, the window gets rotated ten degrees within 200 milliseconds.

As the transformation group can hold any transformation, not only rotations around one axis are possible, but the 3D window can be transformed in any way. Whenever a left mouse click with a modifier key “Ctrl” into the title bar of the window is done, the mouse event is propagated to the parent node in the scene graph.
The Looking Glass environment then catches the event and processes it. The implemented default procedure that is done with a mouse drag with “Ctrl” key modifier is the free rotation of the window. This ability was already implemented by Looking Glass, only the correct event has to be propagated upwards the scene graph tree.



Picture 12 - An example of a free rotation

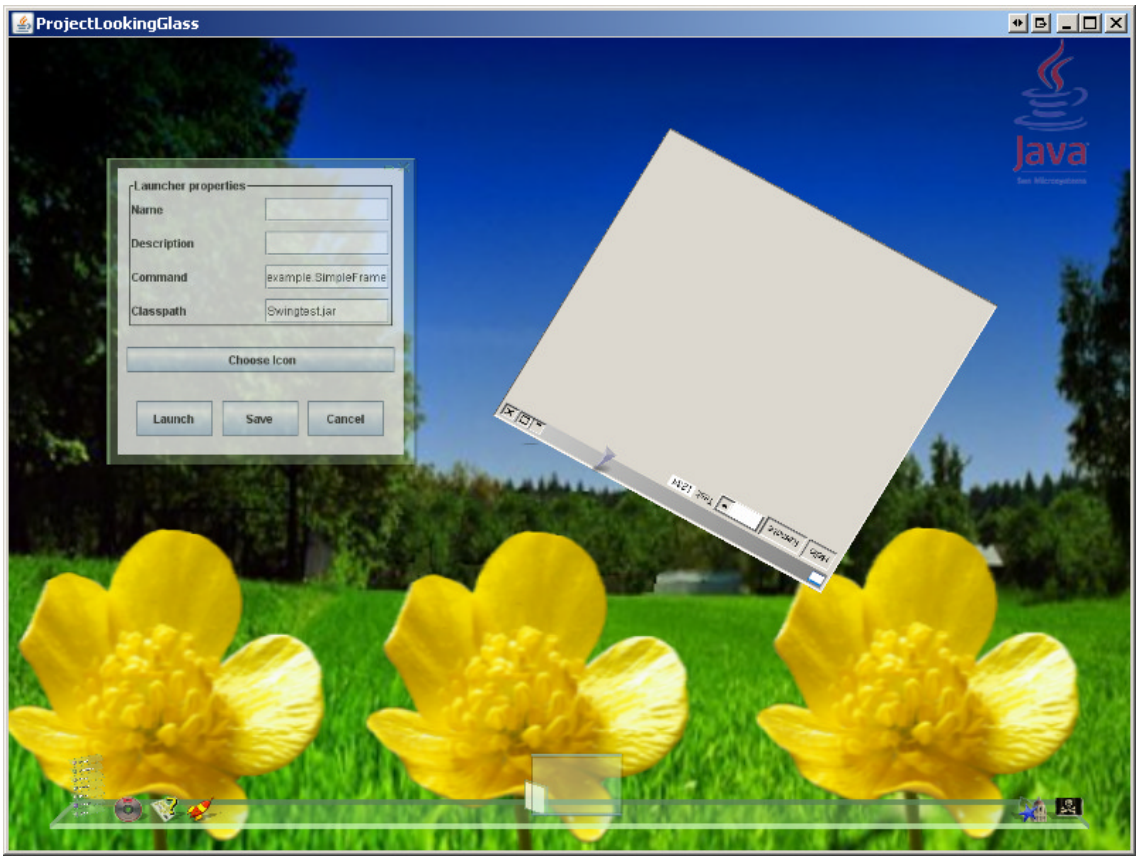
Window resizing

The 3D window within the Looking Glass framework listens for every repaint event of the original 2D SWT application. If a repaint event occurs, the possibly changed picture of the 2D application is mapped to the 3D shape as its new texture. Since the construction (and destruction using the java garbage collector) of the array containing the image data is a very time-complex operation, I added an optimization to the algorithm. Before a new array is created, a quick check is done if the size of the 2D application has changed; if not, the same array can be used to hold the new data, otherwise a new array is created.

As the check for changes of the 2D application's size is done already, the size of the 3D shape is adapted to the new size of the 2D application. This makes it

possible to resize the 3D representative SWT application by changing the 2D application's size.

The events needed to resize the application out of Looking Glass are not yet implemented, but the application is resizable by changing the 2D applications size. This can be done by changing to the 2D SWT application that is run concurrently.



Picture 13 – A resized 3D SWT application (compare to picture 11)

Keyboard events

Looking Glass 3D catches all keyboard events and generates events called *KeyEvent3D.* In the 3D environment there are no selected windows as they are known from the Windows Operating System. Whenever a KeyEvent3D occurs, it is not specified to which window and to which widget this event should be send. Because of this, every SWTWindow3D instance has a field that stores the currently selected widget.

With the help of the stored pointer of the selected widget it was possible to forward a KeyEvent3D by using a Windows message.
Depending on the event type, either “key-down”, “key-up” or “key-typed” the appropriate message is generated:

```
OS.SendMessage(e.getHwnd(), OS.WM_KEYDOWN, e.getWParam(),
                                    e.getLParam());
```

or

```
OS.SendMessage(e.getHwnd(), OS.WM_KEYUP, e.getWParam(),
                                    e.getLParam());
```

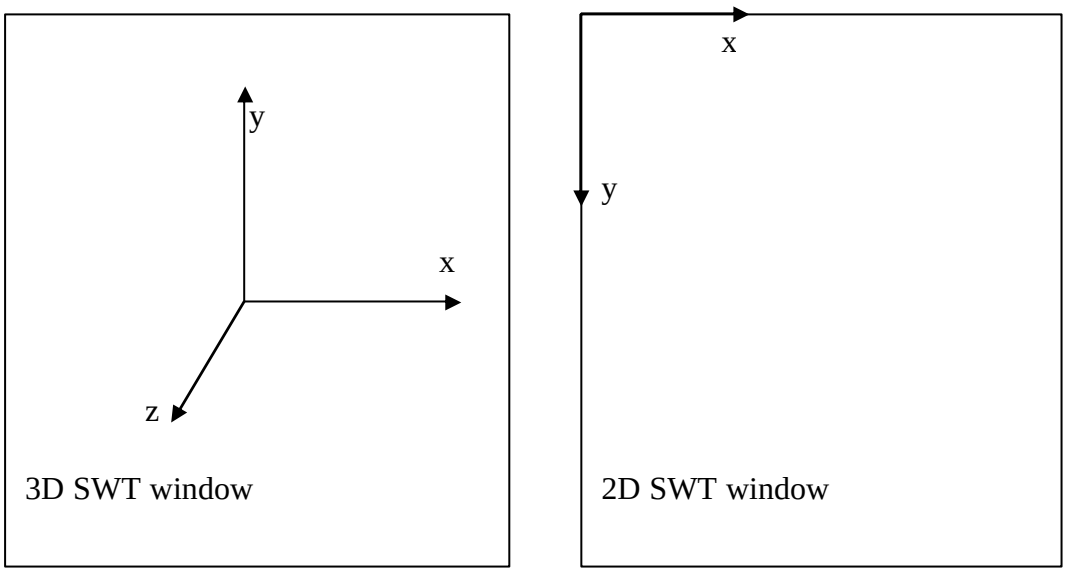
or

```
OS.SendMessage(e.getHwnd(), OS.WM_CHAR, e.getWParam(),
                                    e.getLParam());
```

This Windows message is then received by the Operating System and processed. In some cases a repaint event occurs because of the key event (e.g. if a button was pressed using the keyboard), but in most cases keyboard events are used to type into text boxes and Windows does not generate a repaint when text is typed into a text box.
For this reason, after every key event the texture of the 3D application is forced to be refreshed to show the changes in the text field.

Mouse clicks in general

As the 3D SWT window is just an image of the 2D SWT window I was confronted with the problem that there are no widgets, which can react on the mouse click.
To get the 2D coordinates out of a 3D mouse click the following calculations are done. As the desktop is a 3D area the coordinates of the mouse are given in (x, y, z) format. To get 2D coordinates the intersection point with the underlying object is calculated. At this point, the coordinates of the intersection point are given in global coordinates of the complete 3D scene. To get the coordinates defined in the 3D windows local coordinate system, all transformations down the scene graph tree to the 3D window are applied to the intersection point.



Picture 14 – Coordinate Systems

The calculated local coordinates are defined in a coordinate system, that has its origin in the center of the 3D window, x-axis is pointing upwards, y-axis pointing right. In Windows the coordinate system of a window has its origin in the upper left corner, x-axis pointing right and y-axis pointing down. After the transformation to the local coordinates in the Windows coordinate system, the coordinates are scaled to the size of the 2D window and can be used to find the right widget at the coordinates the mouse click happened.

Looking Glass 3D provides a method to get the local coordinates of the intersection point and just the coordinate system transformation and scaling needed to be implemented. The local coordinates of the intersection point are 2D coordinates, as the z-value always equals zero.

The coordinate system transformation can be broken down to these two lines of code:

```
int x = (int)(3000*lgPoint.x+width/2d);
int y = (int)(-3000*lgPoint.y+height/2d);
```

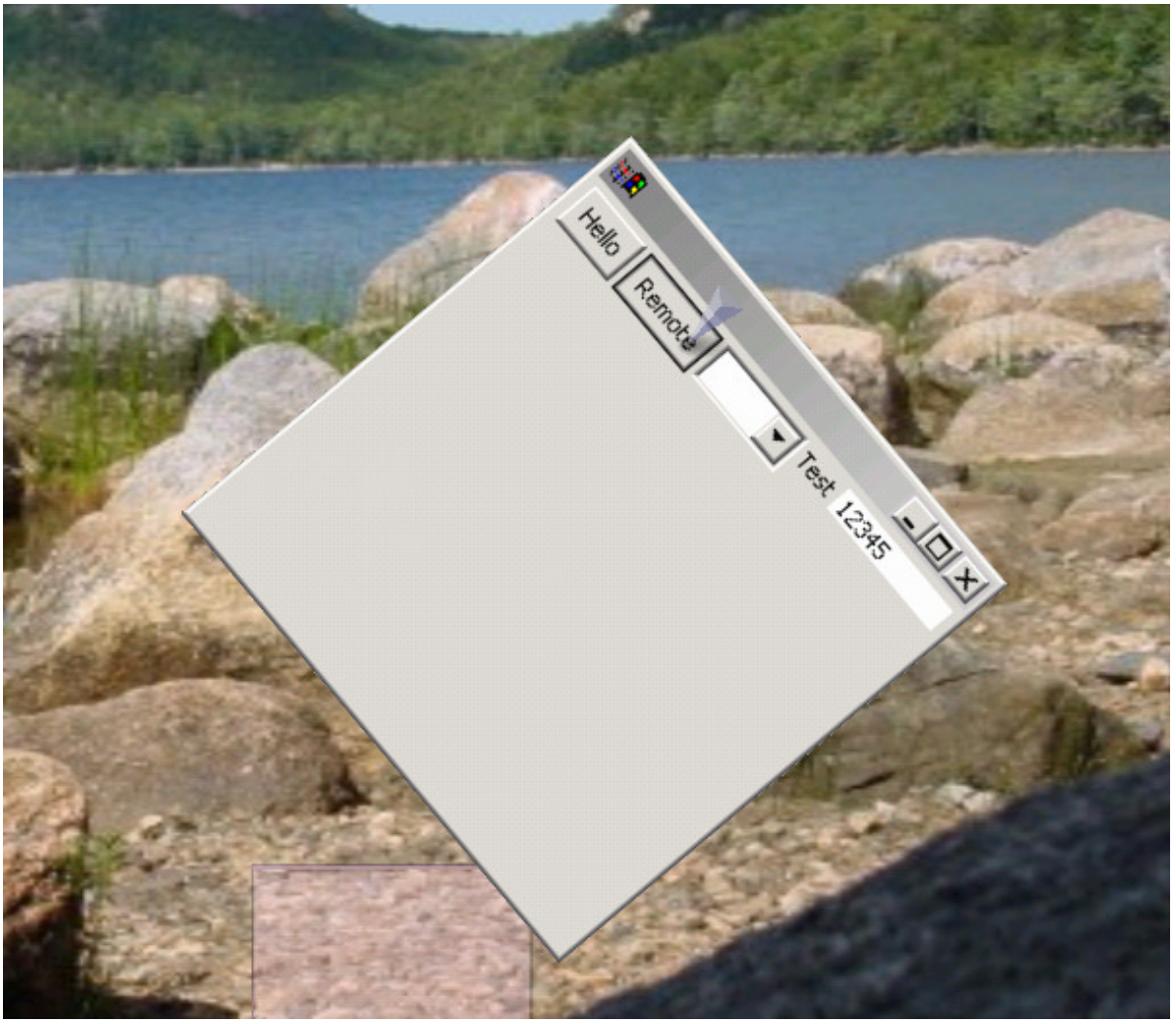
The value *3000* in the calculations is a scaling factor.

Now that I found a way to calculate the coordinates, the widget at this point can be found by iterating through all widgets and test if they contain the calculated

point. As SWT does not allow other threads to interact with the GUI components this is done in a thread executed by the repaint thread of SWT.

Mouse left button single and double click

As the problem of calculating the coordinates was solved correctly and a widget at those coordinates can be found, the implementation of a left button single and double click that can be sent to the 2D SWT window, was an easy thing.



Picture 15 - A left mouse click

When sending the event of a left mouse click to the Operating System several modifier keys can be added to the system message. Looking Glass 3D supports

some of them already and every mouse event contains a list of modifier keys. This list is searched and the associated value added to the system message.

```
for (int i =0;i<modi.length;i++) {
      if (modi[i].equals(InputEvent3D.ModifierId.CTRL))
            wParam |= OS.MK_CONTROL;
      else if (modi[i].equals(InputEvent3D.ModifierId.BUTTON1))
            wParam |= OS.MK_LBUTTON;
      else if (modi[i].equals(InputEvent3D.ModifierId.BUTTON2))
            wParam |= OS.MK_MBUTTON;
      else if (modi[i].equals(InputEvent3D.ModifierId.BUTTON3))
            wParam |= OS.MK_RBUTTON;
      else if (modi[i].equals(InputEvent3D.ModifierId.SHIFT))
            wParam |= OS.MK_SHIFT;
}
```

As the code shows, modifier keys can also be other mouse keys. These are as well added to the parameter sent with the Operating System message.

Mouse right button single click

The right click works the same as the left mouse click. A difference is, that Looking Glass does not support double clicks with the right mouse button yet that is why it is not implemented.

There are some problems with events that mostly follow next to a right click (e.g. pop-up menus); they will be discussed later on. (See “Implementation Problems and their possible solutions”)

Mouse-over-events

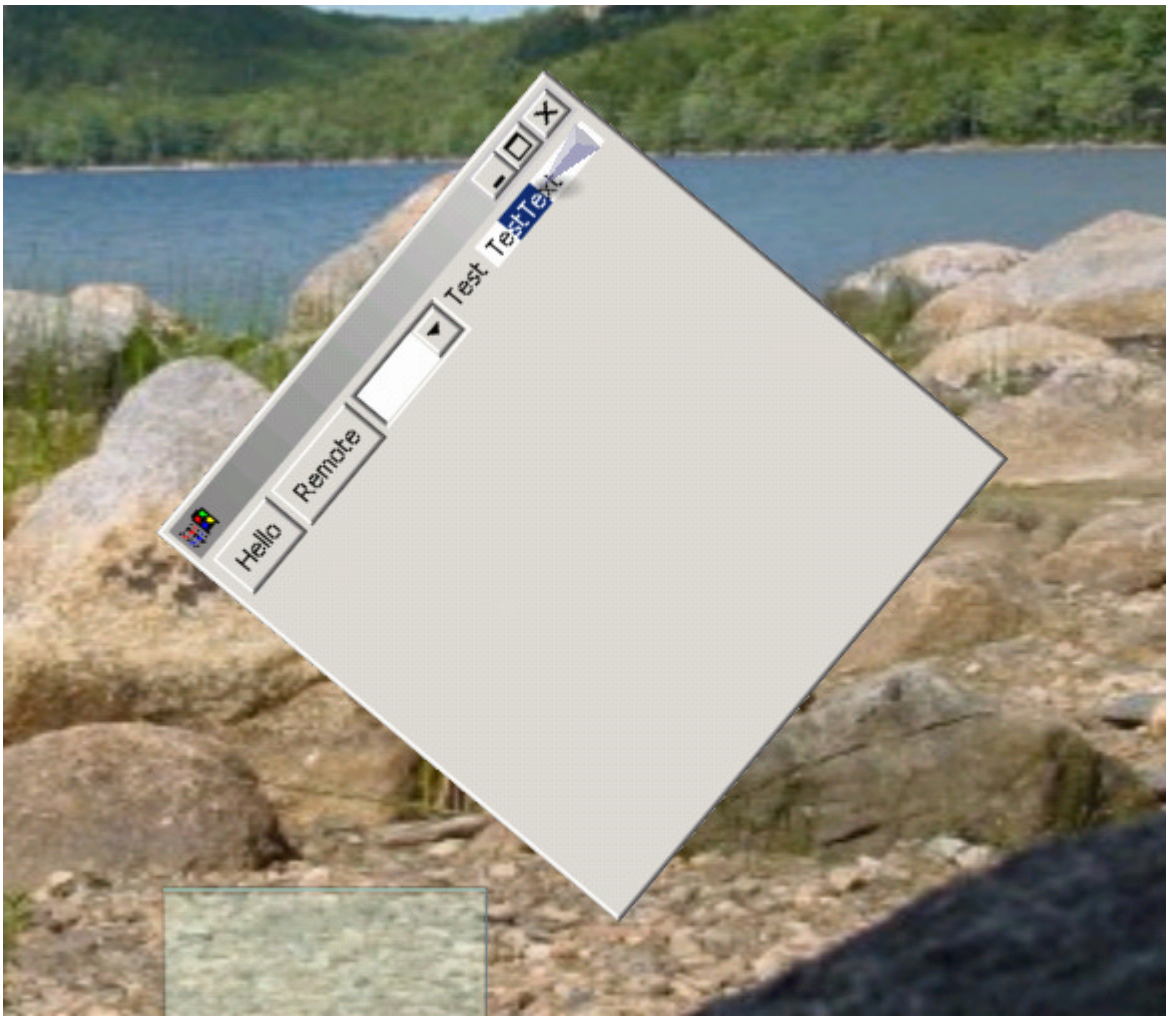
Every time the mouse moves an event is generated that checks if a repaint is needed because of the mouse movement. Examples for such a repaint are tooltips that appear if the mouse is moved above a special widget or widgets that change their appearance/get activated when the mouse moves across them.

As the widget, which is under the mouse pointer must be identified with every mouse move event, the algorithm that identifies the widget needed to be optimized.
The original algorithm iterates through all widgets to identify the correct one. This results in a worst case complexity of O(n), as in the worst case n (the number of widgets) need to be tested. The average number of comparisons is the arithmetic mean, which is n/2 comparisons.
In the optimized algorithm, first a general check whether the widget has changed occurs. The worst case complexity does not change with this performance optimization, but the average case goes down very much.
As in worst case all widgets need to be tested if they contain the mouse coordinates, in average half of the widgets are tested. With a simple check for the same widget that contained the mouse coordinates last time, the worst case goes up by one check, but as most of the time the mouse move does not leave the underlying widget, the average case is near one single check. Of course with the assumption of a normal GUI and not a GUI build out of single pixel widgets.

Mouse dragging

If the mouse is moved while a button is pressed, the movement is called dragging. This event is mostly used to select text within a text field or to drag and drop things within the GUI.
Since Windows does no repaint whenever text is selected, I had to implement an automatic repaint that is done whenever a mouse dragging event occurred. As every mouse movement will then create a redraw event, which captures the

image using java native interface this might become a complexity problem within the algorithm. As the redraw is very time consuming, the redraw is only done if there is no other redraw in progress. I added a semaphore to the redraw method that ensures the method is only run once at a time.



Picture 16 – Text selection

```
public void paintControl(PaintEvent e) {
	if (busy)
		return;
	busy = true;
	refreshTexture();
	refreshTextureAttributes();
	app.setTexture(texture);
	busy = false;
}
```

This simple binary semaphore reduces the overhead of computation time to a minimum. Of course, there is a little possibility that the last redraw that might change the texture in a recognizable way is left out because the busy-flag is set

to true. In reality this case never happens, as the busy flag is only set true if there are too many redraws in little time, this always turns out to be case when the same texture is applied several times.

Implementation problems and their possible solutions

As the goal of this thesis was to implement a prototypical implementation, there are still several problems that must be solved before SWT applications can be run without any problems within the Looking Glass 3D framework.

Off-screen Image capturing

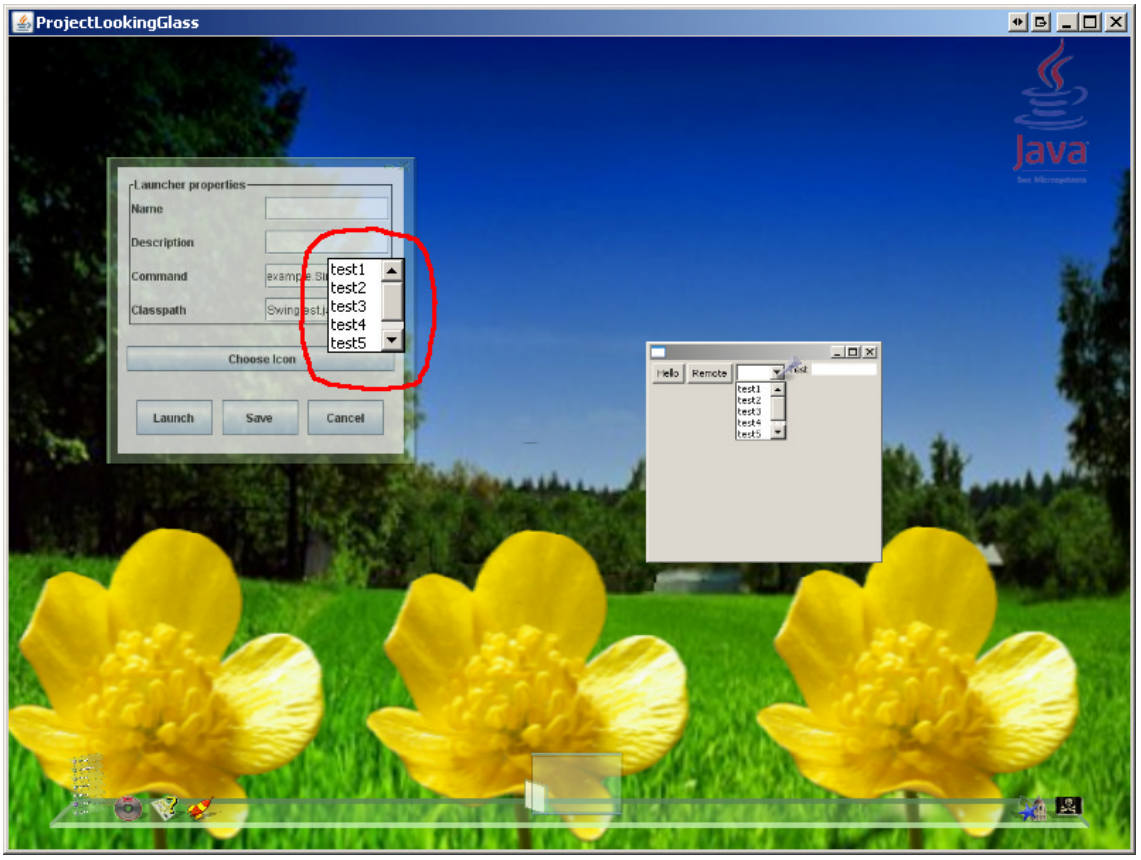
The Off-screen image capturing function only works if the original SWT window is not minimized. Event that are sent to the 2D application while it is minimized are all processed correctly. Even mouse events that interact with the GUI are working (e.g. selecting a text within a textbox). The only thing that is not working while the 2D application is minimized is taking a new image to use it as the new texture of the 3D application.

The reason why it would be good to have everything working while the application is minimized and not only hidden behind the Looking Glass application is that the next problems might disappear if the application is run minimized.

Combo Box menu is always visible

Whenever a combo box opens up, the combo box menu is always on top of everything. This menu is then visible because it is even on top of the active Looking Glass window.

To avoid this error, the SWT application can be moved off the screen or to fix this problem another algorithm must be found that is able to capture the image of a minimized Windows application.



Picture 17 ComboBox menu is always on top

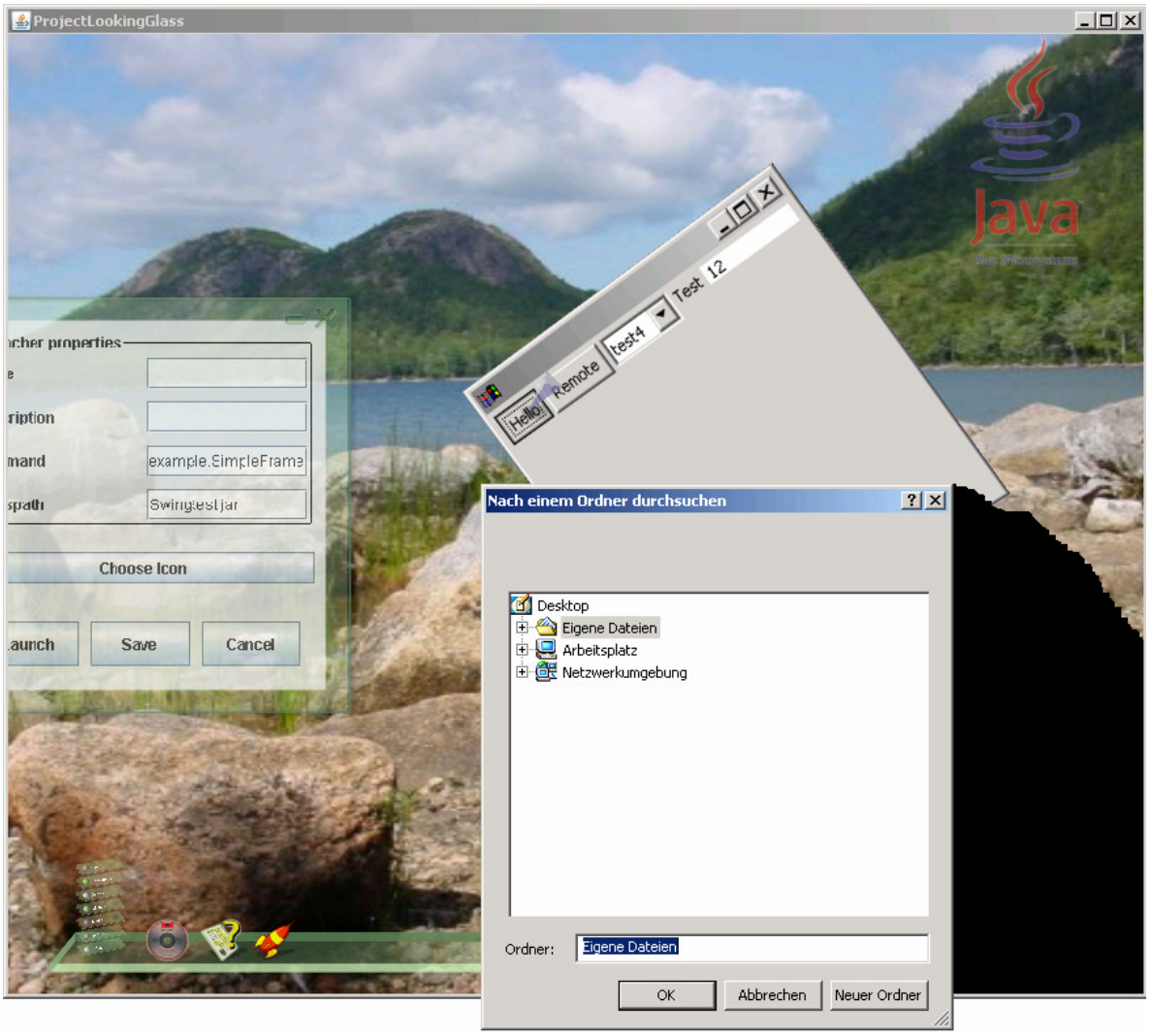
This picture shows the ComboBox-menu that pops up because the 2D SWT application is placed at this position under the Looking Glass framework.

Dialog Boxes

A similar problem as the problem with combo boxes appears when dialog boxes (e.g. file chooser dialog or color chooser dialog) are used.

First of all dialog boxes are no shells, as currently only shells create a 3D representative window, no 3D object is created if a non-shell like a dialog box opens up.

The bigger problem with dialog boxes is that they are modal. That means they block the underlying window until they are closed. Therefore the Looking Glass framework is suspended as soon as a dialog box is opened. Even the repaint function by OpenGL are suspended whenever a dialog box is open.



Picture 18 – The movement of the dialog box shows the suspended redraw (black area)

The focus of input is blocked until the dialog box is closed again. This problem seems to be typical for dialog boxes when they are used out of the Looking Glass framework. SUN has the same problems with their Java Swing support within the Looking Glass framework, that is why they do not support dialog boxes in their implementation.

System Key Events

The KeyEvent3D provided by the Looking Glass environment has the three different types:

- “Pressed”
- “Released”
- “Typed”

The two types “Pressed” and “Released” provide the keys character **and** the keys character code. The type “Typed” does only provide the character.

This problem is handled with this workaround within the SWTKeyEvent class:

```
public int getWParam() {
      if (event.isTyped())
            return event.getKeyChar();
      return event.getKeyCode();
}
```

Since several keys, especially system keys, do not return a character, they do not work with SWT applications running within the Looking Glass framework. Examples for not working keys are backspace and the enter-key.

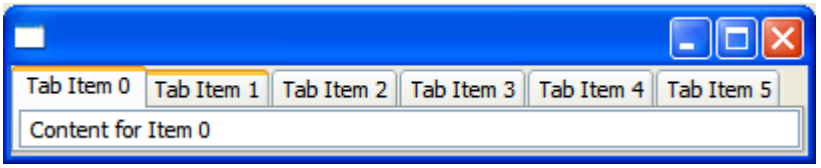
A working solution for this problem might be the re-implementation of the event type “Typed” using the events “Pressed” and “Released”. A “Typed” event is generated whenever a key is pressed and released. “Typed” events are also generated if a key is continually pressed for a certain time.

Stacked Widgets

When a mouse event is received the widget at the mouse cursors position is identified using an algorithm like:

- Iterate through all widgets
- If the widget contains the mouse cursors coordinates return the widget.

This algorithm works correctly as long as there is only one widget at a position. A counter-example where this algorithm does not work can be constructed with the use of a “TabFolder”.



Picture 19 – A “TabFolder” with five “TabItems”

The picture above shows that the content of tab item one has the same region as the content for the other tabs. As any widget in the tabs is a child of the SWT shell, they are all checked for containing the mouse coordinates whenever a mouse click in this shell is generated.
The widget that is found is presumably not the right widget, because it is simply the first widget that is found at these coordinates.

To change this problem, the code would not only have to check whether a widget contains the mouse coordinates, it has to check that the widget is visible as well.
The complexity of the new algorithm will still have time complexity of O(n) as before, but the absolute number of comparisons is increased by one for every existing widget at the same position that is not visible.

Unloading the DLL

At present, only one SWT application can be started within the Looking Glass framework. If another SWT application is started using the provided “Launcher” the following error occurs.

```
WARNING: Executing java app in the same JVM: java example.SimpleFrame
Using ClassLoader org.jdesktop.lg3d.displayserver.LgClassLoader@4adb34

[…]

java.lang.UnsatisfiedLinkError: Native Library E:\Programme\Project
Looking Glass\LG3D Project 1-0-0\bin\ScreenGrabber.dll already loaded
in another classloader
```

Looking Glass starts Java applications in the same JVM, but creates a new ClassLoader for every application. The DLL needed to capture the image of the 2D SWT application is loaded using the static constructor when the first application is loaded. In the moment the second application is loaded, java tries to execute the same static code in another classloader. Because of this, an exception is thrown that the dll-file is already loaded within another classloader.

To solve this problem the best solution would be to integrate the static part of loading the dll-file into the Looking Glass environment, as every created classloader could use the native code that way.

Aliasing Problems in textures

As the texture gets scaled, the image receives some changes that are visible especially in the fonts. In general, a scaled image looses some information, because a picture is constructed out of a finite number of pixels. When the image gets resized not every pixel can be resized.

As mentioned before every picture needs to be scaled up (or down) to the next power of two.
For example the scaling of a picture to four third of its original size will change three pixels to four pixels.

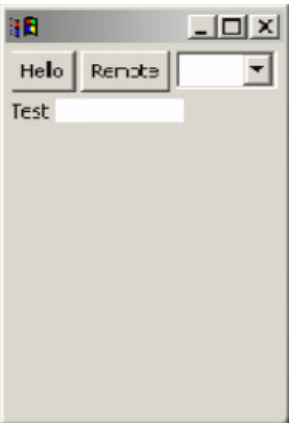
Assuming the three pixels were red, blue and green, then there are several combinations of four pixels that all loose some information to the original image.

- red, red, blue, green
- red, blue, blue, green
- red, blue, green, green
- red, red-blue-interpolated, blue, green
- red, blue, blue-green-interpolated, green

Certainly there are better (and more complicated) algorithms that take the neighbourhood of a pixel into account before its color value is calculated, but they all end up with an image that does not exactly look like the original.

Picture 20 – Original

Picture 21 – 256x256 pixels



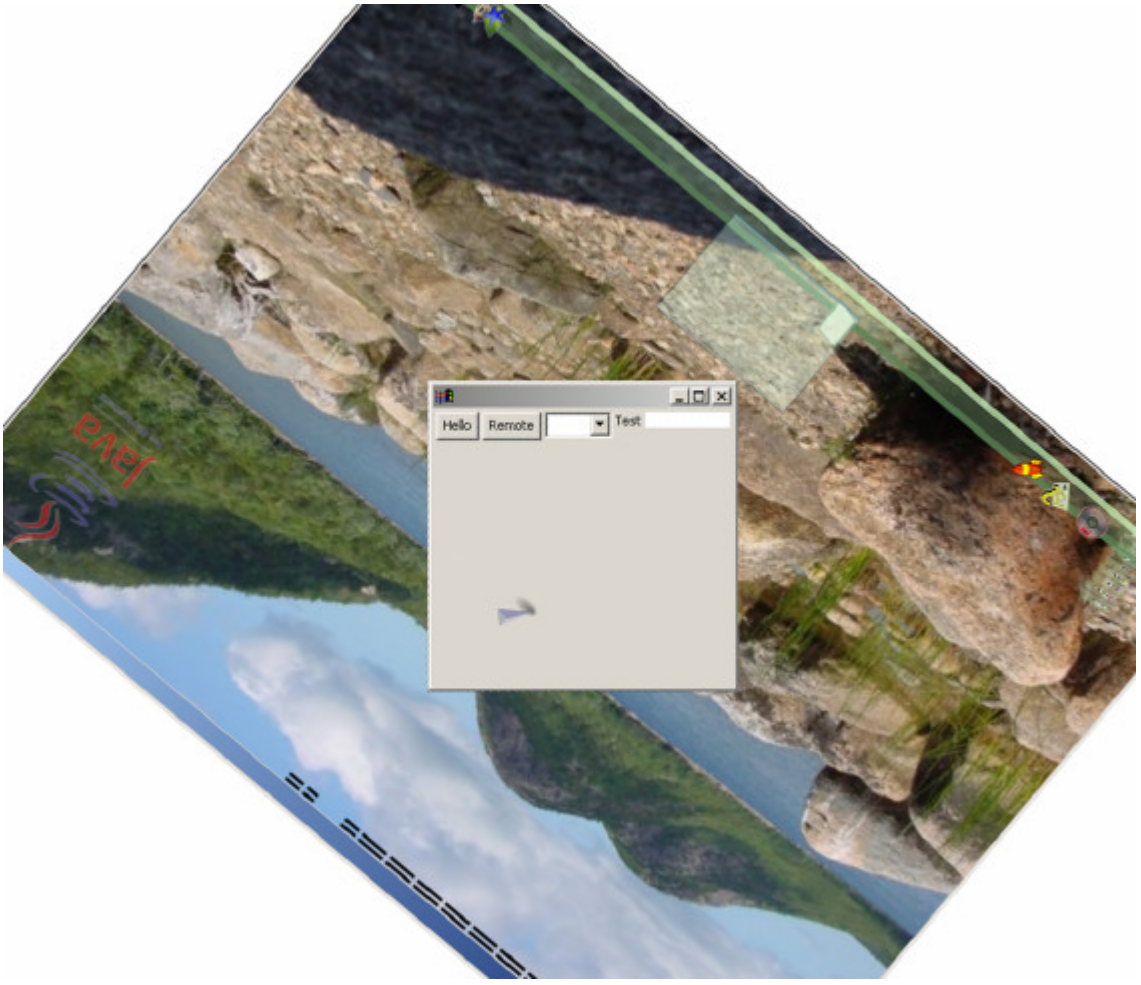
Picture 22 – Texture

The three pictures above show the changes that the scaling did to the original image. The best result would be if the final picture (on the right) still looked like the original picture (on the left).

This problem can be avoided if textures of every side length are used as a texture and not only textures with a side length that matches a power of two. How this could be done is discussed in the Chapter “Look-Out”.

Mouse Input Issues

As long as the application is tested on a normal desktop PC, the user might assume that after being able to rotate the SWT application in any direction, a user at the digital table is able to work from every side of the table. Indead he is able to work from every side, but there is one special challenge to be respected:



Picture 23 – An SWT application rotated towards a user at the digital table

The mouse cursor is of course not rotated yet. Moving the mouse upwards in the picture above will move the mouse pointer to the lower left corner. This is because the mouse coordinate system is defined in the global coordinates of the 3D scene and not within the coordinate system of the rotated window.
As there are more changes to the mouse and keyboard input planned (see chapter “Multiple-Input-Devices”) this problem will be solved there.

Look-Out

The work done until now is the implementation of a prototype that can start SWT applications within the Looking Glass framework. Next to solving the problems described in the chapter before, there are some additional things to do before this project becomes a good SWT framework for digital tables.

Performance improvements

The framework runs fluently on a normal desktop PC, but on a digital table with a desktop resolution of about 8000 by 2000 pixels, the texture handling takes to much time.
The desktop resolution tested with the Looking Glass framework was 3200 by 1200 pixels (each display set to 800x600), at this resolution a full screen application generates a bitmap texture of already more than ten megabytes.
This size of a picture needs to be generated whenever a repaint event occurs.

Power of two texture limitation

The Looking Glass environment is based on Java3D which is based on OpenGL. Every redraw the image of the 2D application is scaled to a texture that has width and height of a power of two. After that the texture is applied to 3D object and gets scaled down to its original size.
The newest OpenGL specification allows the use of textures that do not match the power of two side length constraint. For some reason the graphic card was not able to use the OpenGL feature. Currently, when using textures of arbitrary size the performance goes down as if the system has to render the texture using a software mode and not the hardware accelerated OpenGL mode.

The graphic card is new enough to support the OpenGL feature, an assumption is that there are driver or configuration issues that prevent the card from using the feature. To ensure the graphic card can do textures of arbitrary size, a prototype that tests the feature should be written before buying new graphic cards or testing other drivers.

To create the appropriate texture the class TextureLoader.java provided by Java3D is used, this texture loader does the scaling to the power of two side length. If any texture size would be working this TextureLoader can be avoided, as creating the new TextureLoader instance every redraw is very time consuming.
To support textures of arbitrary size code changes in the class SWTWindow3D would be necessary:

First, the code that stays the same and just creates an image with a reference to the int array that holds the image data:

```
width = size.x;
height = size.y;
ints = new int[width * height]; // the GetDIBits destination buffer
dataBuffer = new DataBufferInt(ints, width * height);
writableRaster = Raster.createPackedRaster(dataBuffer, width, height,
width, bandMasks, null);
image = new BufferedImage(colorModel, writableRaster, true, null);
```

At this position the image that shall become a texture is created and the image data can be changed using the reference to `ints`

```
NativeScreenGrabber.getDeviceBitmap(ints, shell.handle , width,
height);
```

Now the image data is loaded using the java native interface function getDeviceBitmap. The next part is the creation of the texture. In the current code, the TextureLoader is used to create a texture which side length is a power of two.

```
TextureLoader loader = new TextureLoader(image,
TextureLoader.BY_REFERENCE | TextureLoader.Y_UP);
texture = loader.getTexture();
```

If any texture size is supported, the use of the TextureLoader can be avoided and the lines replaced by:.

```
img_comp = new ImageComponent2D(ImageComponent2D.FORMAT_RGBA, image);
```

```
texture = new Texture2D(Texture.BASE_LEVEL, Texture.RGBA,
img_comp.getWidth(), img_comp.getHeight());
texture.setCapability(Texture.ALLOW_IMAGE_WRITE);
texture.setImage(0, img_comp);
```

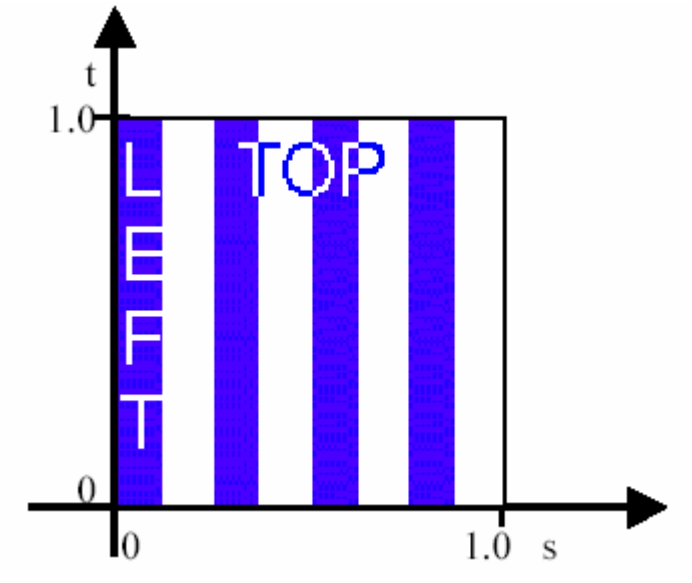
This version sets the image directly to the texture and does not use the TextureLoader anymore. Of course, now the texture has an arbitrary side length.

Individual textures for SWT Widgets

In WindowsXP it is possible to send a repaint event to a single widget. Because of this, it is possible to get an image, which becomes a texture of a single widget. One of the most resource-consuming things is that the image of the application is used as an atomic picture. If only a single textbox changes the picture of the whole application is refreshed.

A great performance improvement would be possible if the large texture of the application were broken up into many smaller textures of the widgets that build the application. To place the smaller textures to the right position the texture coordinates could be used to scale and translate the texture to the right place.

A texture is always specified as a 2D image with a coordinate system that has its origin in the pictures lower left corner and scaled the width and height to one.



Picture 24 A texture coordinate system

To map the texture to a 3D shape a bijection is defined, that maps one 2D texture coordinate to one 3D coordinate. Using the right texture coordinates the textures for every single widget can be added to the shape individually and splits down the high memory usage of a large texture that shows the whole application. The smaller textures will not reduce the overall memory usage, but as every newly added texture is transferred though the rendering pipeline inside the graphic card, less data has to be transferred when a smaller texture is exchanged.

Multiple Input Devices

A digital table with the ability to rotate every window to every side is still unhandy if it can only be operated by one person at a time. To add the ability, that more than one person at a time can interact with the digital table, first of all concurrent input devices must be implemented. Subsequently the implementation of using these multiple input devices within the rotating windows can be approached.

As the rotation of SWT windows is not implemented in the Operating System itself, but implemented within the virtual desktop environment Looking Glass, the multiple input device feature, that is necessary for a digital table must be implemented in Looking Glass as well.
The implementation of multiple mice and multiple keyboards is probably even easier within the Looking Glass Framework than directly within Windows (or any other Operating System).
The reason for that is that Windows cannot move application windows concurrently as there is no dispatcher handling concurrent input events from different devices. Within the Looking Glass framework, a dispatcher can be implemented that reacts directly on the input events.

Multiple Mice

A mouse pointer in Looking Glass is nothing else than a 3D object that listens to mouse events and behaves like a normal mouse. To add another mouse pointer to the Looking Glass environment, another 3D shape that reacts on another mouse input needs to be created.

In Windows two windows may not be active at the same time. In the Looking Glass framework, a window is a 3D shape that is covered by the image of the 2D application; this does not prevent the two “windows” to move concurrently, as 3D shapes can move concurrently, as long as a dispatcher is implemented in Looking Glass to receive the move events concurrently.

The Concurrent Input Framework, developed by Andreas Herbig, provides Multiple Mice for SWT applications in Windows XP. This framework uses a shell to simulate a mouse cursor. This shell is always on top of the other applications and reacts on the mouse input.

The basic idea of this framework can be used to add more mice to the Looking Glass framework if the shell, representing the mouse cursor, is exchanged with a 3D shape.

The advantage of using Looking Glass as the platform to support multiple mice is the simplicity with which events can be generated. In Windows XP the mouse pointer needs to create the correct Windows System Messages, but in Looking Glass, the mouse pointer just needs to create the right-, left-, middle- or wheel-mouse-event and the framework takes care of creating the correct Windows Message as this feature had to be implemented to support SWT windows anyway.

Multiple Keyboards

Next to the convenience of multiple mice comes the comfort of multiple keyboards. The concurrent input using multiple keyboards is again easier implemented in Looking Glass than in Windows itself.

To send correct key events, the current implementation has the workaround of storing the selected widget (mentioned in chapter “Implemented Features”) and using that pointer to send the events.

With multiple keyboards, a global Storage of selected widgets is needed. This storage is a one to one relation of keyboards to selected widgets and whenever a key event of a specific keyboard is received, the correct widget is looked up and receives the event.

As every keyboard has an associated selected widget, a problem becomes obvious:

One Keyboard and one mouse must become a pair, because the mouse selects the widget, to which the keyboard can send events. This must be remembered when implementing multiple input devices, otherwise the system cannot find the correct focus for the keyboard input.

In Windows it looks as if keyboards only can send events to selected widgets. This looks like a problem for multiple keyboards, because there is only one selection at a time. But this assumption is wrong, as the selection (e.g. a blinking cursor) is nothing else than a visible feedback for the user, which widget is currently selected. Using the Operating System messages *WM_KEYUP*, *WM_KEYDOWN* and *WM_CHAR* it is possible to send Keyboard events to any widget without having to select the widget.

Conclusion

The Feasibility Study and Prototypical Implementation of a Window Framework for Digital Tables was a very interesting project as a Diploma thesis.
Now in retrospect the produced code looks very easy, you might think this is doable within two weeks without any problems.
However, the real aim of this thesis was and outcome is the feasibility part, identifying on the one hand the dead ends and on the other hand the most promising possibilities to start further work from.

The implementation of a framework that supports rotation for digital tables directly within MS Windows XP or Mac OS X is nearly impossible. Both Operating Systems are not Open Source and both Operation Systems need many changes within the window manager.
The Linux Operating System does not have this problem, as it is Open Source. Nevertheless, the changes that need to be done are so much work that Linux disqualifies itself as well as the other Operating Systems.

Therefore, I identified the implementation in a virtual environment as the best solution. The system I chose was SUN's Looking Glass 3D, where the implementation is much easier than in Linux e.g. A reason for that are the existing OpenGL features like transformations and clipping against other objects.
In my opinion, the implementation within an existing 3D framework is the best solution because half of the work is already done. As the complete framework for the digital table does not only include the rotation of windows, the decision if or if not everything should be implemented within a virtual environment cannot be made without knowing the problems of the other features.
The virtual environment Looking Glass provides an easier API than the Operating Systems, especially when considering the topic of multiple input devices. All operating systems (Mac OS X, Windows and Linux) are designed

for one keyboard and one mouse; this makes it very difficult to implement more keyboards and/or mice directly within the Operating System.
Looking at the global picture that a complete framework for digital tables has to be implemented, it was important to find an environment that supports as much as possible designated features and provides the API to implement the rest of them.

The implementation of or in a virtual environment is completely different to the implementation directly within the Operating System.
Although changing an Operating System was the first idea when I started this thesis, all tested Operating Systems caused so much trouble that putting more effort into the development within the Operating System made no sense any more.
The implementation within an existing virtual environment like SUN's Looking Glass 3D is much easier than the implementation in Linux (which was left as the only other choice). A reason for that are the existing OpenGL features like transformations and clipping against other objects.

In my opinion, the implementation within an existing 3D framework is the best solution because half of the work is already done. As the complete framework for the digital table does not only include the rotation of windows, the decision if or if not everything should be implemented within a virtual environment cannot be made without knowing the problems of the other features.
The virtual environment Looking Glass provides an easier API than the Operating System, especially when looking into the topic of multiple input devices. All Operating Systems (Mac OS X, Windows and Linux) are designed for one keyboard and one mouse; this makes it very difficult to implement more keyboards and/or mice. Looking Glass on the other hand does not rely on the Operating Systems mouse cursor or keyboard input.
If any other input device is connected to the PC, the Operating System can be ignored and the device can be used as needed within Looking Glass.

Looking at the global picture that a complete framework for digital tables has to be implemented, Since Looking Glass is already Java based and is therefore an easier platform for Java SWT applications than any Operating System it is the current best solution for a complete framework for digital tables.
Since Looking Glass is already Java-based and is therefore an easier platform for Java SWT applications than any Operating System, it is the current best solution for a complete framework for digital tables.

After identifying SUN's Looking Glass as the platform I wanted to work with, all other occurring problems where problems related with the implementation details. The visible features like rotating, resizing or moving the 3D window were all implemented using the API of Looking Glass, that is why these features where implemented very easily.
The most complicated problems like sending the correct system events or capturing the image of the 2D application were all Operating System related.
This confirms my decision of choosing a virtual environment.

The source code of this project is available at:
http://mase.svn.sourceforge.net/viewvc/mase/DigitalTable/lg3d-swt

Appendix

Abbreviations and Glossary

LCD - Liquid crystal display

"A **liquid crystal display** (LCD) is a thin, flat display device made up of any number of color or monochrome pixels arrayed in front of a light source or reflector. It is prized by engineers because it uses very small amounts of electric power, and is therefore suitable for use in battery-powered electronic devices."[1]

SWT - Standard Widget Toolkit

"The **Standard Widget Toolkit** (SWT) is a graphical widget toolkit for the Java platform. It is an alternative to the AWT and Swing Java GUI toolkits provided by Sun Microsystems as part of the Java standard."[2]

OS - Operating system

"An **operating system** (**OS**) is a computer program that manages the hardware and software resources of a computer. At the foundation of all system software, the OS performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking, and managing files. It also may provide a graphical user interface for higher level functions. It forms a platform for other software."[3]

[1] http://en.wikipedia.org/wiki/Liquid_crystal_display
[2] http://en.wikipedia.org/wiki/Standard_Widget_Toolkit
[3] http://en.wikipedia.org/wiki/Operating_system

DLL- Dynamic-link library

“Dynamic-link library (also written without the hyphen), or **DLL**, is Microsoft's implementation of the shared library concept in the Microsoft Windows and OS/2 operating systems. These libraries usually have the file extension DLL, OCX (for libraries containing ActiveX controls), or DRV (for legacy system drivers).

The file formats for DLLs are the same as for Windows EXE files — that is, Portable Executable (PE) for 32-bit Windows, and New Executable (NE) for 16-bit Windows. As with EXEs, DLLs can contain code, data, and resources, in any combination.

In the broader sense of the term, any data file with the same file format can be called a *resource DLL*. Examples of such DLLs include *icon libraries*, sometimes having the extension ICL, and font files, having the extensions FON and FOT.”[1]

GUI - Graphical user interface

„A **graphical user interface** (or **GUI**, often pronounced "gooey"), is a particular case of user interface for interacting with a computer which employs graphical images and widgets in addition to text to represent the information and actions available to the user. Usually the actions are performed through direct manipulation of the graphical elements. “[2]

JNI - Java Native Interface

“The **Java Native Interface** (**JNI**) is a programming framework that allows Java code running in the Java virtual machine (VM) to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages, such as C, C++ and assembly.”[3]

[1] http://en.wikipedia.org/wiki/Dynamic-link_library
[2] http://en.wikipedia.org/wiki/GUI
[3] http://en.wikipedia.org/wiki/JNI

JVM - Java Virtual Machine

“A **Java Virtual Machine** (**JVM**), is a **virtual machine** that interprets and executes Java bytecode. This code is most often generated by Java language compilers, although the JVM can also be targeted by compilers of other languages. JVM's may be developed by other companies as long as they adhere to the JVM standard published by Sun.”[1]

I/O - Input/output

“In computing, **input/output**, or I/O, is the collection of interfaces that different functional units (sub-systems) of an information processing system use to communicate with each other, or the signals (information) sent through those interfaces. Inputs are the signals received by the unit, and outputs are the signals sent from it. The term can also be used as part of an action; to "do I/O" is to perform an input or output operation.”[2]

X - X Window System

“In computing, the **X Window System** (commonly **X11** or **X**) is a networking and display protocol which provides windowing on bitmap displays. It provides the standard toolkit and protocol to build graphical user interfaces (GUIs) on Unix, Unix-like operating systems, and OpenVMS, and is supported by almost all other modern operating systems.”[3]

[1] http://en.wikipedia.org/wiki/JVM
[2] http://en.wikipedia.org/wiki/Input/output
[3] http://en.wikipedia.org/wiki/X_server

API - Application programming interface

“An **application programming interface** (API) is a source code interface that a computer system or program library provides in order to support requests for services to be made of it by a computer program. […]The term API is used in two related senses:

- A coherent interface consisting of several classes or several sets of related functions or procedures.
- A single entry point such as a method, function or procedure.”[1]

GTK+ - GIMP Toolkit

“The **GIMP Toolkit**—abbreviated, and almost exclusively known, as **GTK+**—is one of the two most popular widget toolkits for the X Window System for creating graphical user interfaces. GTK+ and Qt have supplanted Motif, previously the most widely used X widget toolkit.

GTK+ was initially created for the GNU Image Manipulation Program, a raster graphics editor, in 1997 by Spencer Kimball, Peter Mattis, and Josh MacDonald—all of whom were members of eXperimental Computing Facility (XCF) at UC Berkeley. Licensed under the LGPL, GTK+ is free (and open source) software, and is part of the GNU Project. “[2]

GDK - GIMP Drawing Kit

“**GDK** (GIMP Drawing Kit) is a computer graphics library that acts as a wrapper around the low-level drawing and windowing functions provided by the underlying graphics system. Originally developed on the X Window System, GDK lies between the X server and the GTK+ library, handling basic rendering

[1] http://en.wikipedia.org/wiki/API
[2] http://en.wikipedia.org/wiki/GTK

such as drawing primitives, raster graphics (bitmaps), cursors, fonts, as well as window events and drag-and-drop functionality."[1]

Widget

"In computer programming, a **widget** (or **control**) is an interface element that a computer user interacts with, such as a window or a text box. Widgets are sometimes qualified as *virtual* to distinguish them from their physical counterparts, e.g. *virtual* buttons that can be clicked with a mouse cursor, vs. physical buttons that can be pressed with a finger. Widgets are often packaged together in widget toolkits. Programmers use widgets to build graphical user interfaces (GUIs)."[2]

Shell

"Instances of this class represent the "windows" which the desktop or "window manager" is managing. Instances that do not have a parent (that is, they are built using the constructor, which takes a Display as the argument) are described as *top level* shells. Instances that do have a parent are described as *secondary* or *dialog* shells."[3]

[1] http://en.wikipedia.org/wiki/GDK
[2] http://en.wikipedia.org/wiki/Widget_%28computing%29
[3] http://www.eclipse.org Class Shell API documentation

Sources

[1] “Java(TM) Native Interface: Programmer's Guide and Specification”
by Sheng Liang, available at http://java.sun.com/docs/books/jni/

[2] MSDN library
http://msdn.microsoft.com/windowsxp

[3] The GIMP Toolkit GTK+
Source code available at www.gtk.org

[4] Mac OS X Website
http://www.apple.com/macosx

[5] “Window Contents Capturing using WM_PRINT Message“
by Feng Yuan (author of Windows Graphics Programming: Win32 GDI and DirectDraw, http://www.fengyuan.com).

[6]Java3d Project home
https://java3d.dev.java.net

[7] lg3d Project home
https://lg3d.dev.java.net

[8] Wikipedia – The free encyclopedia
http://www.wikipedia.org

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in dieser oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegen.

_____

(Sascha Hömig)