Executive Summary

This thesis describes the process of building an interface that allows users to create 3D graphics using Google SketchUp on a system called PiVOT. PiVOT is a multi-view touchscreen tabletop display. Interfacing it with SketchUp gives users the advantage of seeing different views of the 3D model on the same screen. The project is split into three main steps – displaying the SketchUp viewport in the 'Shared View', displaying the rendered models in the 'Personal View' and integrating the touchscreen.

1. The term 'Shared View' is used to refer to the view observed by users when they are sat down in a chair in front of PiVOT. The SketchUp viewport is the working environment of the application where the models are created and edited using different tools. A projector is used to display the Shared View, so the first step of this project is to generate a buffer of images of the SketchUp viewport. These images are calibrated prior to projecting them so that they appear on the screen in the correct dimensions.

2. The 'Personal View' is the view observed when the user leans above the tabletop. To generate this view the models developed in SketchUp are exported and then imported into the rendering engine which renders them with the corresponding light and shading settings. This buffer of rendered model images is displayed on the same screen as the Shared View.

3. The third step of this project is receiving commands by touch on the screen and sending them to the model rendering application shown in the Personal View or to the SketchUp application shown in the Shared View.

Main Contributions:

1. **Designed the interface according to the steps presented above; implemented the interface by connecting the hardware and software elements together to work, communicate with each other and transfer data successfully.**

2. **Implemented the touchscreen interaction in the PiVOT system and its interaction with the SketchUp 3D models via the interface.**

3. **Demonstrated that the interface is functional and accurate enough to use the same model in real-time interactive applications**

Achievements:

- Designed the software pipeline as a result of researching different implementation options and choosing the appropriate hardware and software products
- Projected the SketchUp models on the screen as describes in the design
- Displayed rendered models exported form SketchUp on the LCD panel
- Achieved near real-time update of the rendered models after they have been modified in SketchUp
- Integrated the application in the PiVOT software system using the PiVOT API
- Proved the use of the interface model in the development of other applications by looking into the functionality of the interface together with its evaluation

Contents

Chapter 1

Introduction

1.1 Motivation

The popularity of animated movies is growing rapidly nowadays. Whole studios are engaged in the creation of animations and thousands of animators are being hired to do the character and set design and modeling. In the animation production pipeline, this modeling is preceded by drawing the scenes, objects and characters in the movie. The drawings are being discussed and if needed redrawn before the process of creating models starts. The animation is then developed based on the approved drawings.

This thesis is motivated by this mapping of a drawing to a 3D character and set. Artists who create these models need to constantly go back to the original 2D image and reproduce it in 3D. To do this, they need both the 2D and 3D views to be close to each other and easy to compare. The faster the animator can compare the two views the faster he can assess the development of the artwork based on the second view and edit the model. The increase of speed of this process decreases the loss of focus experienced by this animator and enhances the creativity.

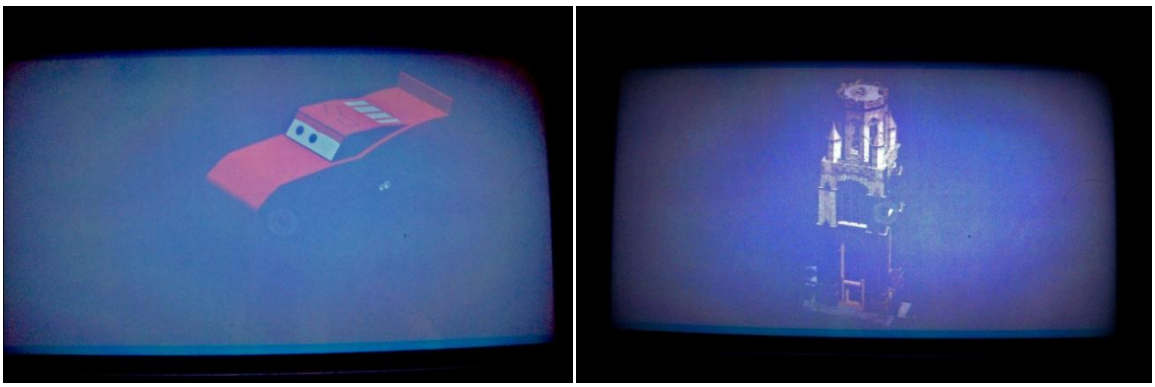


Figure 1: A rendered view of Pixar's Lightning McQueen (left) and rendered view of Wills Memorial Building (right) displayed on the multi-view display

Creative practices such as sketching, character animation and other usually require users to work with specialist software packages. Most of these packages require the user to switch between two or more views –one to create content, another to view the content and another one to view geometric or other information about the product. The purpose of these views is to enable better control on the content creation process. This means users constantly need to switch between these views to see if their content is fit for the purpose. This slows the creative process down which can lead to loss of focus or inspiration. An example of such a software product is Google SketchUp. SketchUp is a 3D graphics modeling software which is used by a lot of animators, architects and regular users. They work with SketchUp develop their models using a lot of different views – top, bottom, rendered, etc. They often need to switch from one to another view window to be able to modify the object from different sides. This is not a very convenient method to use the information from different windows. Today's display systems address these problems by connecting more than one monitor to a machine, which can be expensive and space consuming. With the advent of novel tabletop systems like PiVOT it is now possible to provide users with multiple views on the same display surface without any user interaction with the device.

1.2 PiVOT –Personalized View-Overlays for Tabletops

PiVOT is a newly developed multi-view tabletop display system designed by the Bristol University Interaction and Graphics Lab. This system has the capability to present users situated at different positions around the tabletop with personalized views. It can display several different images on the same screen. One of these images can be viewed by a group of people situated around the tabletop. The second image can be observed by those leaning over the screen. These views are displayed on the same screen using see through techniques.

Figure 2: A user of the interface as she is rotating the ball rendered on the screen

1.3 Project Aim and Objectives

The aim of this project is to create an interface to work with Google SketchUp modeling software on the PiVOT tabletop display. This interface introduces a way of animating that gives users the advantage of seeing different views of the model on the same screen.

Main Objectives

- Identify the input and output parameters for the display system and modeling software
- Design the control and communications pipeline between the different devices.
- Review different software products and choose the solution that fits best the particular part of the design
- Choose the appropriate hardware
- Write scripts and plugins to control the software and organise its communication with the tabletop
- Implement the interaction of the touchscreen and 3D models through the interface
- Test the system, analyse the performance of the interface and its ability to support real-time modeling applications

1.4 Thesis Organisation

A review of approaches and solutions related to these objectives is presented in the rest of this report. Chapter 2 will present a summary of research made in similar technologies, present the PiVOT system and its software in depth. An overview of SketchUp is given in this section. Chapter 3 gives more detailed information about the design of the interface and Chapter 4 describes its implementation. Chapter 5 presents the results from experiments made on the interface and evaluates its performance. Chapter 6 discusses the future work that can be done in the area.

Chapter 2

Background

2. 1 Related Work

2.1.1 Stereoscopic multi-view displays

In the 1960's the first head worn see-through displays were invented. Their original purpose was for users to be able to see information that is displayed on a screen along with the physical objects behind the screen. Since then a lot of research has been made to develop similar display systems. Some of the implementations do not require any devices to be worn by the user. DigiScope (1) is one such device that allows for users to see objects behind a transparent screen. It uses a special film that either blocks the light or lets it pass through it depending on the angle it is coming from. In this way the objects behind the screen and the information on the glass-pane can be both seen.

There are displays that have been developed to provide stereoscopic imaging, without the need of glasses. Such a display was presented by Ye, State and Fuchs in their paper (2). Other similar devices are MUSTARD (3) and Illusion Hole (4) which make use of blocking masks to accomplish the multi-view functionality. They allow for multiple users around a tabletop to be able to see different images. This is achieved using masks which block the image one person receives from the other viewers. Only this user can see the view displayed on this overlay. and doesn't allow this person to see other people's personal views. Stereoscopic devices supporting 3D content creation have been developed for many purposes. Such a purpose is the use of this tabletop in medicine. In (5) the advantages of glasses-free 3D for medical imaging are discussed. The paper specifies the operation of a multi-autostereoscopic system.

2.1.2 Digital Art

A lot of research has gone into developing software that can simulate the creation of some kind of art. The application "EverybodyLovesSketch" (6) aids new as well as experienced artists to create 3D curve sketches based on gesture capture. Another idea was put through by Animaatiokone (7), which used a specially created hand-driven device to create stop-motion animation. Conte - (8) a small interactive device with the shape of a crayon, recreates the crayon strokes on a tabletop display.

2.2 PiVOT

PiVOT is a tabletop system developed by the Bristol University Interaction and Graphics lab. It has the capability to display more than one view on the same screen.

2.2.1 Overview of System Features

PiVOT supports two views - Personal View and Shared View. The Shared View is generated by a projector displaying images on the screen. A special film places on this screen makes it possible for this view to be visible only from a certain direction. The personal View, on the other hand, is produced on the LCD screen and is visible only when leaning above the screen and looking down. It has the functionality to display models in 3D using head tracking and motion parallax techniques.

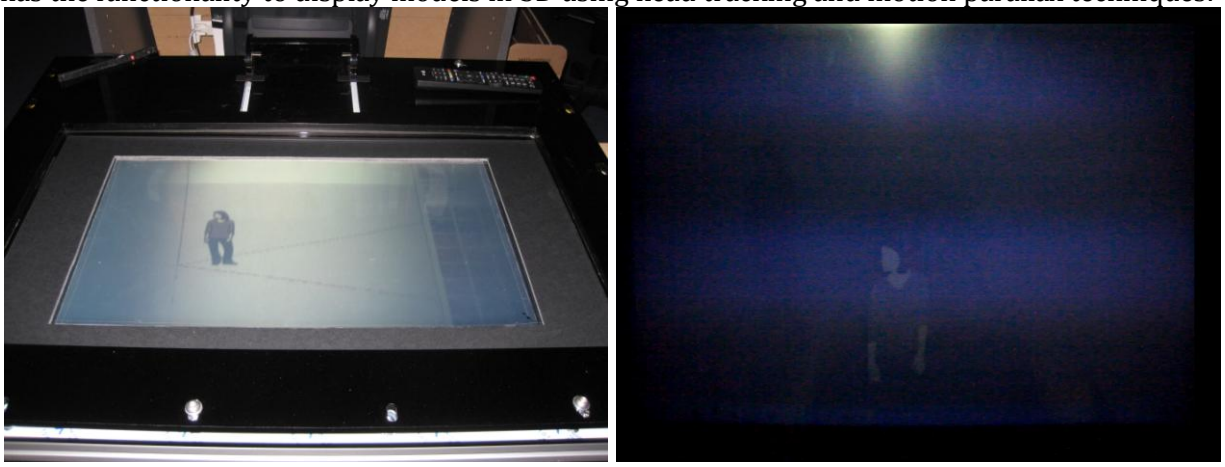


Figure 3: Display of the system; only the Shared View is visible from this angle (left), the Personal View is visible when observed directly from the top

As shown in Figure 3, when situated in the Shared View zone, one cannot see the models in the Personal View. Similarly, if views are looking at the screen from above they are outside the Shared View zone and can only see the Personal View. Moving the head back and forth gives the ability of a user to swiftly transition from one view to another.

2.2.2 Hardware Components

The PiVOT system consists of the following devices:

- projector to display the Shared View
- 2 Liquid Crystal Display panels to create the Personal View
- optical tracking camera - used to generate the monoscopic 3D
- touchscreen – new addition to the system
- lumisty film - diffuses the light from the projector in one direction

2 computers connected with a cross-over cable

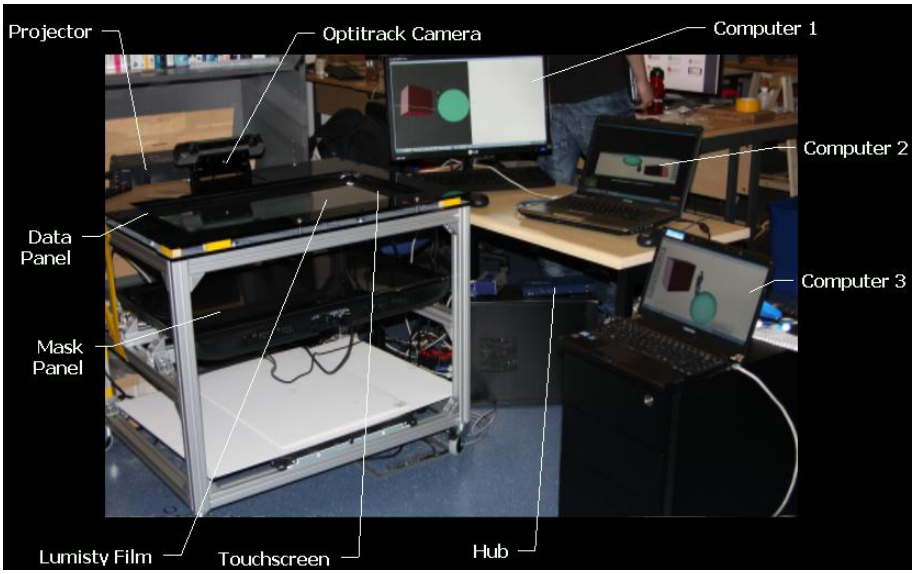
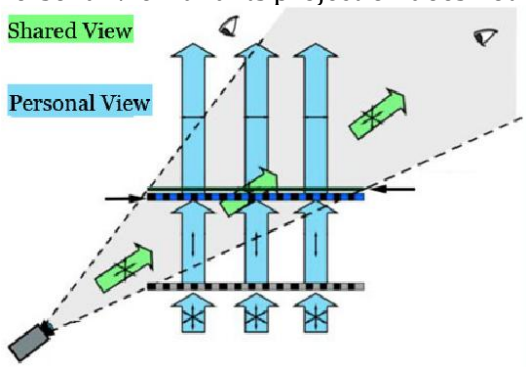


Figure 4: The components of the PiVOT display system

PiVOT consists of two separate display systems – one for each of the two views. The Personal View is created using two liquid crystal displays taken from LG LCD Televisions. The two displays are positioned one over the other at a distance of 10cm and form a structure resembling a "LC sandwich". The upper screen displays the personal overlays for the different users. The multi-view property of PiVOT works on the same principle as in MUSTARD –using a hole mask to display multiple views.

The second view - Shared View, is produced using a projector situated behind the LCD panels and projecting through the 10cm gap between them. The Shared View is created separately from the Personal View and its projection does not conflict with the images displayed on the data panel.



<u>Figure 5:</u> The Shared View can be observed at the angle at which the projector light is directed (green arrows). The Personal View can be observed directly from the top (blue arrows)

There is one addition to this piece that allows both views to be displayed on the same screen. This is achieved by a film layer positioned on top of the upper LC display. When a user leans forward he is able to see through this layer and the personal view on the upper LC panel becomes visible to him. Users further apart cannot see through the film and observe only the shared view produced by the

projector.

This see-through film layer used is called Lumisty film. This film works in the following way- when viewed from a certain angle it is transparent, but when observed facing the viewer it is not. It therefore diffuses the light coming at it at a certain angle. There are different Lumisty films – one type – MFX is clear when viewed from the side and frosted when viewed from above. MFY – a second type shows the opposite characteristics. When the projector light hits this diffusive surface its image becomes visible. If the projection hits the Lumisty film in an angle different than the diffusive angle, the light will pass through this film and the image will not be visible.

Furthermore, even when the projector is positioned at a correct angle to the film that diffuses light, the viewer needs to be located in a certain zone to be able to see the shared view. The reason for this is that the Lumisty film does not diffuse light in every direction. The user needs to be situated in both the film diffuse zone and projector zone. In the projector zone observers see the projected image, but not the data on the LC sandwich. In the see-through zone, viewers can see the image displayed on the LC sandwich but do not see the projected view. In this zone the viewer is able to see through the Lumisty film at the LC display sandwich and sees the personal overlay.
The PiVOT system provides mixed content for the personal overlay. An observer can acquire either 3D or 2D view. This feature is implemented with the use of different hole mask configurations. The position of the personal overlay image can be controlled by special markers tracked by the system. A user can also move the marker to change the position of his personal view or touch the screen to interact with it.

The personal overlay view is achieved using 2 LC displays ("LC sandwich"). It is made out of two monitors. One of the LC displays acts as the data panel and has its rear polariser removed. The second screen acts as the active element necessary to generate a dynamic mask. PiVOT works on the principle of polarization of light. First of all, it is able to simultaneously display different images if viewed from different sides due to the hole mask situated on the lower of the two displays. The light that passes through these holes is polarized by the display and when it hits the second polarized display, the image shown on it becomes visible to the eye. Only light that passed through the holes of the mask gets projected on the upper display and makes the pixel colors visible. The part of the display that is not lit from the panel below does not display any information. This results in viewers from different positions being able to see different patches of the image. In order for them to be able to see the other patches of the image the hole mask needs to be rotated. To show the whole image the mask needs to be rotated to different positions to make rays of polarized light pass through all pixels of the upper display. Over a short period of time the colors of all pixels will become visible and the user. Finally, a fast rotation of these holes on the mask will create the effect of seeing all parts of the display almost at the same time, creating the effect of observing a whole image.

This is the technique one uses to see different images from different viewpoints – the technology behind the multi-view functionality of Mustard and consequently PiVOT is based on polarization of light in different directions. When light is projected from the lower LC display, it gets polarized in a certain direction. It then reaches the second LC display and depending on its direction of polarization, it either passes through the LCD or gets blocked. If the data layer polarizer is oriented in the same direction as the first one the polarized light will simply go through and no image will become visible. On the other hand, if this polarizer is placed perpendicular to the first one, the light coming from below will get blocked and make the color of pixels on the display visible.

Every pixel on the upper display is associated with a 3 values. These values represents angle at

which the red, blue and green components of the light get displaced when the polarized ray from the second panel hits that pixel. If one of the displacement values is 0 then the light ray component associated with this value passes through the display without making the corresponding color visible. On the other hand if the displacement is 90 degrees, the ray wave becomes perpendicular to the display, gets blocked by it and makes its color visible.

Projector

The projector used for the display of a shared view – a NEC WT610 short-throw model, projects light through the 10cm gap between the displays in the sandwich. Its light can pass only through the data panel out of the two panels, because it this display cannot block the projector's non-polarised light. Finally, the Lumisty film on top of the data panel diffuses the projected light and the projection becomes visible on the screen. If it was put underneath the upper display it would depolarize the light coming through the lower display.

Figure 6: The projected illuminated the upper display panel by projecting through the 10 cm gap between that panel and the lower mask panel

Another feature of the system is its ability to track the viewer's eyes and in this way manipulate the 3D content displayed on the screen. This is achieved using a camera to track the movement of a person or his marker. Finally, the touchscreen is a new addition to PiVOT.

Touchscreen

The touchscreen was added to the device at the start of the project. It is placed on top of the data panel and connected to the Shared View node through a USB port.

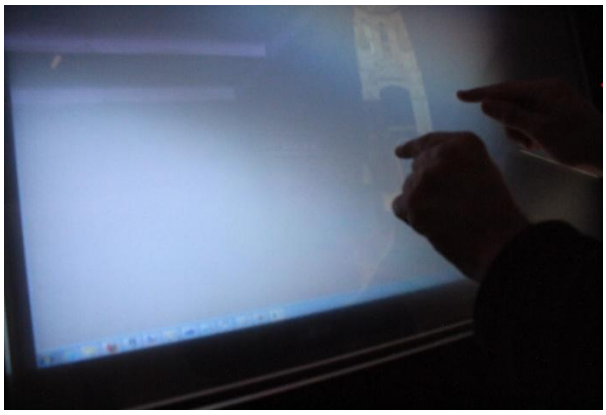


Figure 7: A user interacting with the model of Wills Memorial Building

Camera

- Camera for tracking - Grasshopper Express GX-FW-10K3M-C Optitrack camera, 1024x768 resolution, 60 fps capture speed.

This device is responsible for finding the position of a person's eyes using a tracker that is placed on his head. The hardware arrangement of the system is displayed on the diagram on the right. The arrangement of the panels is as follows Lumisty film of type MF-Y is put on top of the data display. The projector is situated below the panels and projects light up, though the 10sm gap. Two computers connected via an Ethernet switch are used to produce the shared and personal views. One of these machines is responsible for the projected image, and the other computer () is responsible for manipulating the mask and data displays:

Shared View Node

- Machine responsible for personal overlay - Dell Precision 7500 – Intel Xeon E5620 processor; 6GB DDR3 RAM, NVIDIA Quadro FX 4800

Personal View Node

- Machine responsible for the Shared view - Dell Inspiron 620 (Intel core i-3 2100 processor, 4GB DDR3 RAM: NVIDIA GeForce 9500GT graphics card, running Windows 7
- LCD sandwich - 2 displays taken from LG IPS2321P 23, 1980x1080 monitors

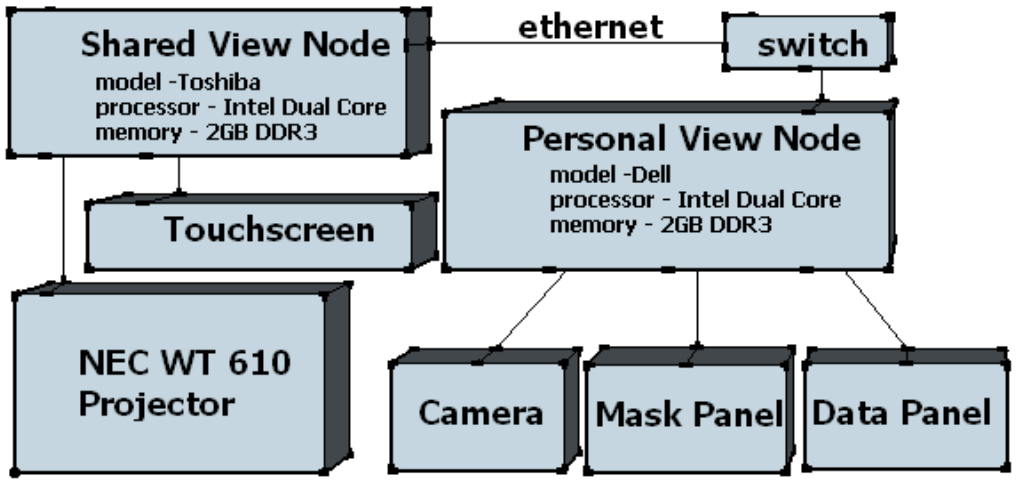


Figure 8: Hardware arrangement and connections between the devices

2.2.3 Software Components

The software working on PiVOT has several elements with different functionality. The system API and some small applications are built using C++. The operating system running on the machines is Windows 7.

Ogre

The product that renders the models to be displayed in 3D on the tabletop is called Ogre. Ogre is an object-oriented rendering engine used by a large number of game and other modeling application developers because it provides extensive support for building games, simulations and other interactive 3D applications. OGRE can be used to work with external libraries and different input devices and allows the developers to add whatever sound or other physics they choose. Ogre works together with some 3D implementation libraries (OpenGL, DirectX). This interface eases the process or rendering and is designed to be abstract from the Direct3D and OpenGL libraries. This section gives a brief introduction in the way the Ogre environment is set and the main objects that are used to create and render it.

Ogre Scene

The Ogre scene is the 3D environment that is rendered using object oriented programming. The objects in this scene can be meshes, terrains, lights and other particles. Meshes are objects made from triangles which have a texture of material assigned to them. The main object in Ogre is the Root. Root is responsible for arranging the whole scene to be rendered. Access to other classes in the system can be gained and instances of these classes can be manipulated using this object. The Root can create Scene Nodes. These objects are part of the scene graph used to organise the objects in the scene. Scene Nodes can create Child Nodes. Objects can be attached to both the Scene and Child Nodes. Another type of object - the most common type, is the Entity object. It is an instance of type mesh and one or more materials can be assigned to it. Entities can be attached to Scene nodes and moved together with them. A material can represent a texture, color or a combination of the two with different characteristics in terms of lighting and shadow receiving (shaders). A texture can be made from any image loaded into Ogre.

Resources

The ogre resources can be divided into different object types - Mesh, Texture, Material. A resource can be loaded separately or as a member of a resource group. The resource group consists of .mesh, .material and texture files. Textures can be in the form of PNG, JPEG, TGA , ZIP types. They are images attached to a mesh using a material object they correspond to.

The resources loading in Ogre is divided into five steps and the stages each resource object goes through are four. The first of these states is the 'Unknown' state during which the filename is stored in the Resource Group but Ogre is unaware of its existence. When a resource group is created resource locations are added to it. The next step is to declare the resource and its type to Ogre after which the resource file transitions to a 'Declared' state. Further on, the Resource Group is initialised which creates the declared resources and parses the scripts located in the resource group. The resource passes to the 'Created' stage. Finally, the resource is loaded into Ogre by creating an entity from the .mesh file, by loading the resource using a load function or by loading the whole resource group directory the file belongs to. At this moment the resource passes to the 'Loaded' state. In order to reload the resources, a reload function is called on the resource of the whole resource group. This function first unloads the resource and puts it back in the 'Created' state and then loads it back into the 'Loaded' state. This procedure updates all the .mesh files but does not parse any material scripts. Therefore, in order to reload the textures and materials too, the resource group

must be cleared, which returns the resources to an 'Unknown' state and re-initialise it in order to apply the necessary changes to the material scripts. When a group is initialised, all its material and texture scripts get parsed. In automatic resource loading all resource groups are initialised by Ogre at the same time. If manual resource loading is chosen, then the groups are initialised one by one by the user.

Rendering

The Root node has one more important task. It takes care of the continuous scene rendering by calling the function StartRenerding(). The rendering loop ends only when the application exits or when FrameListener objects stop it in order to perform tasks in between the frame renderings. One such listener is the FrameRenderingQueued. It is a loop which executes between every two frame renderings of the Main Ogre thread. Mouse and key press events can be captured during that loop and all objects in the scene including the scene nodes can be modified. The modified objects are then rendered at the next iteration of the rendering loop in the Ogre Main thread. Two different programming interfaces can be used to render the 3D models. They are:
Direct3D is a renderer subset of DirectX, supported by Windows only
OpenGL - Open Graphics Language, is an interface which is supported cross-platform. The OpenGL rendering can be called using most programming languages. It is designed to provide hardware-accelerated rendering. Using this rendered does not support well the manual allocating of resources. On the other hand, Direct3D is gives users the freedom to manage their own hardware resources.

Libraries

The following libraries are used to develop the PiVOT software framework:
- Ogre has a lot of useful features such as flexible plugin architecture. Using it new functionality and connection with other products can be achieved. In the case with PiVOT, OGRE communicates with the camera that tracks the movement of a person takes care of rendering the exact views of the 3D objects that need to be passed further on along the pipeline (to NVIDIA?) to display on the screen. For the personal overlay view, the retrieved images are two – one for each of the left and right eyes. The two images are renders of the 3D setting in OGRE made from two cameras positioned at viewpoints in the 3D working space. The coordinates of these two cameras are input from the tracking camera with the
use of ARToolkitPlus. OGRE is also used to generate the masks and their projections on the upper LC display.

- Finally, OGRE integrates with NVidia by passing the images needed to create the stereo effect. These two images are merged together using NVidia Cg programming. This is a high-level shading language. It is used for programming vertex and pixel shaders and graphics processing units.

- ARToolKitPlus is one of those libraries which extends ARToolKit library. It is used to track physical markers in 3D. Depending on their coordinates and orientation it is able to calculate the position and orientation of the camera. This library is used to track the personal overlay markers (also called "flings") on the tabletop. The position of the markers determines where the personal overlay window will be situated. ARToolKitPlus is also used to find the relative position of the eyes of a person viewing the tabletop and the orientation of his head. At the moment this is accomplished using 2 markers placed on the head of the user. The calculations and the updates of the user's position are made in real-time.

- NaturalPoint tracking tools and API written as C++ functions are used to capture the images from the Optitrack camera that does the tracking of users and the markers they are wearing.

- OpenCV is used to pre-process the images captured from the camera, in order to improve the detection of ARToolkitPlus. Histogram equalization is performed – adjusting the contrast of the images using their own histograms. Another technique implemented in this software is adaptive tiled thresholding which uses different thresholds for different parts of the image.

- Finally, OIS is used to manage keyboard/mouse input and to communicate with OGRE rendering operations. OIS is an object oriented input library. It is an open source library which makes it flexible to be extended and cooperate with other programs. In the device describe here OIS provides support for input data from a keyboard and mouse, but generally it can also operate with joysticks. To capture input data the OIS needs to be initialised and mouse and keyboards objects created. (These objects can be buffered or unbuffered.) There are several methods (keyPressed, mouseMoved, etc.) that describe various events. They are only called when an event is generated.

PiVOT API

Apart from the elements used in the operation of PiVOT described above a special API is developed to aid the development of applications that operate on the system. It allows programmers to do several things -gather information from the real world, switch the operation mode -random hole mask, vertical parallax, mono, etc. PiVOT defines a base class to represent an application so that developers can create new case studies by inheriting it. The interface of this base class describes several methods. The main ones are the following:

- Scene creation: This method is called by the PiVOT framework during the startup of the system. In this method, the contents that PiVOT will display (the virtual 3D world) is created.

- frame sent to the GPU: This means that OGRE has just sent the frame to be rendered by the GPU. This process is heavy and leaves the processor almost idle, so it is a good moment to perform any heavy-duty computing that your application requires (AI algorithms, etc). Usually this method is the entry point for the logic of your application. It is called once every frame.

- frame rendering finished: This means that OGRE has finished rendering the current frame. Only light computations here (time stamping, profiling, etc.)

The software framework developed for PiVOT has 3 operation modes. It eases the process of application development with. The software can operate in monocular mode which is generally used to create 2D and non-stereo motion parallax 3D content. The input is only one image for both eyes. Motion parallax can provide information about the depth at which an object in a scene is positioned. It represents the difference of the distances between stationary objects which is subject to change over time as the observer moves. The background moves slower than objects that are close by, which give information about how far they are. Motion parallax algorithms are implemented in the stereo display of images on PiVOT. When a user moves, the images in the foreground of the displayed scene move faster than those in the background, which creates the feeling of depth of the view. VPB and RH masks and are used to display stereo 3D content.

2.3 SketchUp - software for modeling of 3D objects

SketchUp is an application developed by Google to create 3D models. It can be installed and used on MAC OS X and Windows operating systems. Once the installation is complete users can create their own models or load models from the SketchUp Warehouse (database of various 3D models).

SketchUp also has applications not only in character animation but also architectural, game, mechanical design. It is very intuitive and easy to use which allows beginner in art or design to quickly learn to create 3D objects. SketchUp is a free software and has recently become more and more popular both with amateurs and professional model and design developers. Its increasing popularity and level of difficulty make it suitable for developed interface.
[Expand, 3D comic books]

2.3.1 SketchUp Viewport

The viewport, together with its instruments is organised in the way show in Figure 4. The camera is situated to give a perspective view of the objects being created. Users can add a background image and create their model by reshaping and resizing 3D objects placed in front of them. A set of tools is available to work on the created models such as extrusion, eraser, fog and shadows adding. The perspective can be set to be at an angle with all the axes or perpendicular to the plane made of two of these axes. A large set of colors and different textures (rock, wood, roof) are available when setting a material to a model.

2.3.2 Exporting Models from SketchUp

Plugin development for SketchUp is based on ruby scripts. It is easy to add more functionality to the application or automate certain drawing, painting or shape creating tasks. Using ruby, the user interface can be modified to include special commands in the SketchUp Tools menu.

There exists one SketchUp to Ogre exporter to convert SketchUp files (.skp) to Ogre files (.mesh). It works in the following way :

The SketchUp to ogre exporter work in the following way also shown in Figure 9:
- It creates structures to store the geometry of the entities the model to be exported is made from.
- Name/rename materials, textures and meshes, deleting any spaces
- Export the geometry, export the materials and finally - the textures.

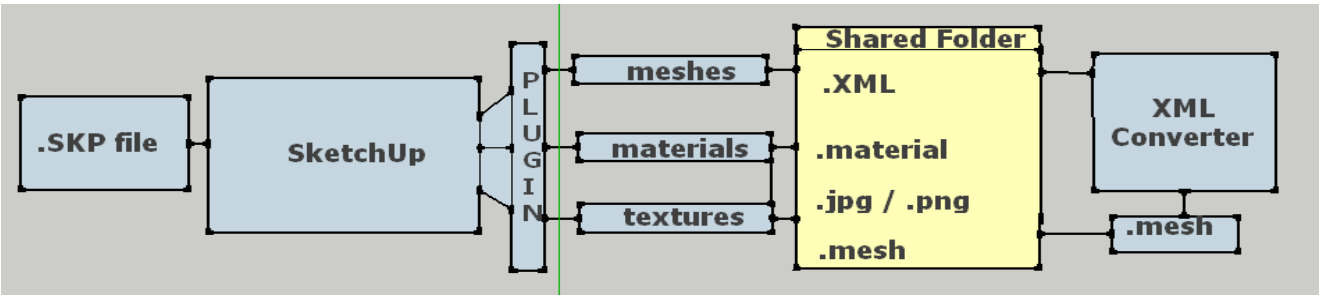


Figure 9: Procedure for exporting meshes from SketchUp; the plugin exports materials and textures separately to the shared folder along with an XML file. This file is converted into a mesh by the XML converter and saved in the same shred folder, ready to be imported by Ogre

Chapter 3

Design

3.1 Theoretical Approach

Implementing the interface starts with finding all the components fit for the purpose of the project and creating the necessary connections between them. To accomplish the final result different design decisions were made. In cases of hardware or software constraints choices for using other products were made.

Use of the hardware and software in the system to achieve the project aim
The ability of the tabletop display to produce two different views without any occlusion or disappearance of data from the screen is the system's main advantage. The Personal and the Shared View are used to build the interface, as they both present different information about the object being modeled. The Shared View is used by the artist to create a model in SketchUp ad make changes to it when required. The Personal View on the other hand shows the rendering of the model. Having both views on the same screen enables the user to receive different information by moving his head back and forth. If a change is made in the SketchUp model, this change will be reflected in the rendering and will become visible to any user that is leaning over the screen. Leaving the Shared View zone means the only view he can see on the screen at that position is the Personal View. Vice versa, if the user is in the Shared View zone he cannot see the Personal View. The artist can thus focus on building a model and editing it. Switching between the two views will take logr This can be done without losing focus. This comes as a result of the smooth transition between the two views.

3.2 Behavior of the Interface

The results to be achieved are a system which enables the user to create surfaces, curves and meshes by interacting with the tabletop. Users are expected to see the 3D image of their models when they lean above the table. Changing the model in the Shared View will result in its change in the Personal View.

3.3 Software and Hardware Products

The design of the interface before beginning its actual implementation is crucial. A lot of different software products with similar functionality exist. They can achieve certain functionality for the system have particular features which means they have different limitations. Some pieces of software can not work together even though they may achieve very good result on their own. A lot of design decisions were made, depending on the advantages and disadvantages of different products. The main requirements are for the software products to be compatible with the PiVOT system environment - Windows 7.

Design decisions had to be made concerning the necessary hardware along with the software requirements. The way the software pipeline for the interface is designed gives certain constraints on using hardware devices. The Personal View needs to be displayed on the LCD sandwich from a computer running Ogre with the rendered models. A second computer is needed to run another instance of Ogre where the SketchUp viewport is rendered and reshaped according to the calibration. This Ogre render window is projected as the Shared View. A characteristic the Ogre render window is that it updates its content only if it is focus. Therefore, the two Ogre processes need to run on separate computers and remain in focus. Furthermore, SketchUp is also designed to update only if the window is in focus which gives two design solutions. The first one is to use a script to control automate the switching of windows in and out of focus. A BASIC-like scripting language called Autoit provides this functionality to the Windows GUI. Using it will force the SketchUp and Ogre Render window to exchange places in the arrangement of the screen and constantly change the window in focus to allow update of both applications. Another more elegant solution that will not run a script and create stalls is using a third computer to run the SketchUp application. This method gives the freedom to run the three main processes without any competition for the CPU or memory which improves the quality of the rendered view and improves the performance of the interface in terms of speed. For this design, therefore, a hub is needed to connect the three computers in a network.

3.4 Design of the interface

In short, the projected image is shown on the screen in the corresponding shape. ThinVNC is used to capture the viewport. It captures the SketchUp screen and loads it into a browser. The browser content is then rendered in OGRE. The image is calibrated by mapping it to a polygon shape in OGRE with precomputed coordinates. The resulting image is sent to the projector.)

3.4.1 Shared View - choice of products

The Shared View is a projection of the SketchUp viewport. The viewport needs to be captured and projected on to the lumisty film on the screen. Before projecting it every frame needs to be preprocessed. This is necessary because the projector is situated behind the LCD panels and the view is projected at an angle. The projected images therefore need to be calibrated.

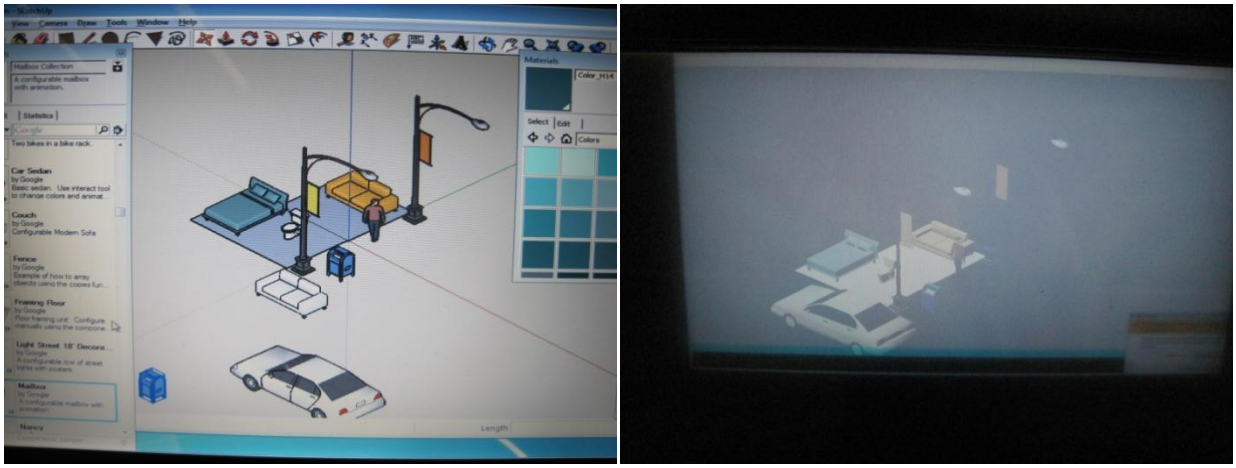


Figure 10: The Shared View is a capture of the SketchUp viewport (left), the Shared View does not conflict with the Personal View (right).

VNC

To be able to project the viewport onto the projector in needs to be captured and rendered in Ogre. Furthermore, the design aims at realising the update of the SketchUp viewport as users pass commands using the touchscreen. For this purpose remote desktop sharing must be used.
Most programs on the internet that achieve desktop sharing are based on Virtual Network Computing (VNC) system. It allows the remote control of the desktop of another computer with the same system installed on it. VNC works by creating a framebuffer and transmits raw pixel data over the network.

Several software products that provide this functionality for operating systems including Windows 7 and are free to use but the public exist. Most of them implement the remote framebuffer protocol to access the graphical user interface of the remote computer. This makes them similar to the VNC system. Similar products are TeamViewer, Crossloop, RealVNC.

All of the above are small screen sharing applications with similar features but they all need additional plugins to be able to share the screen capture through a web browser. ThinVNC is the most suitable product because it makes use of any web browser without additional plugins. It does not require any additional installation on the client side which makes it suitable for the use in a system comprised of many software components. ThinVNC allows screen sharing and remote control with the mouse. It provides the necessary functionality by only installing it on Bob and MM does not need to install the application. The service operates by transmitting JSON and JPEG image encodings of the remote screen via HTTP protocol. Unlike RealVNC, the users of ThinVNC do not need to set up the port forwarding. The only thing that needs to be changed is the service authentication settings. ThinVNC provided the option of userID and password authentication. This option must be changed so that the service requires no authentication when ran in the Ogre application.

Embed the Web Browser Content in Ogre

A couple of libraries can be used together with Ogre to enable rendering of HTML content. They use the image of the web page to drawn a texture, which can be used by Ogre in to assign to an entity or manipulate in another way.
Awesomium and Berkelium libraries use Chromium (open source part of Chrome) to load a web page. Berkelium is an open source library of large size. Its advantage over Awesomium is that the code can be modified to fit the project needs. However, for this project a straightforward implementation of web content rendering and less build time is needed. Awesomium is closed source developed by a company and has an easy to use API which makes it suitable to use as a black box in the overall application. It allows for web content to be rendered on a 3D object in Ogre using the C++ interface and allows interaction with the webpage by injecting button pressed and mouse events into the browser. Awesomium constantly updates the structure that holds web content and is able to modify the texture that corresponds to it. This means that when the web page changes the rendering of the object on Ogre will be also modifies. Consequently, when the screen capture of SketchUp is loaded into the web browser any changes in the viewport will reflect in the texture and the object this texture is assigned to. Finally, through the awesomium library function interaction with SketchUp can be realised. Mouse and key events received in Ogre can be injected into the web browser, put through to the remote desktop by ThinVNC and reflected in SketchUp.

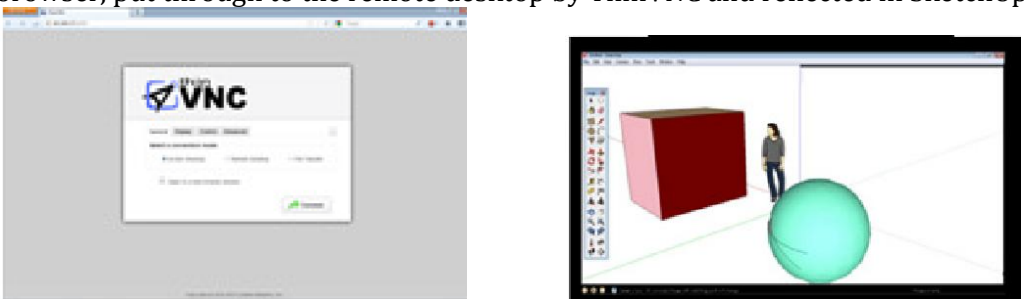


Figure 11: The SketchUp viewport (right) is captured by VNC and embedded into Ogre as web browser content(left)

In this implementation the user interaction with Ogre is designed to come not from the mouse and keyboard but from the touchscreen. This design allows for the touch on the screen to control the creation of squares, cubes, spheres and the assigning of colors, materials and specific textures to these objects. Users have the ability to choose the tools and instruments for model development and viewing by making a touch on the PiVOT screen.

Calibration

After the SketchUp viewport is captured its shape needs to be changes before projecting it on the screen. This distortion should be in a way that will make its projection fit the required screen area. Every captured image of the viewport is saved a texture in Ogre and applied to a polygon of the required shape. Setting the required dimensions of this polygon is called calibration.

Input: Image to be projected
- Load image as a texture by passing it to the graphics card

- Use a prior calibration to define coordinates of the input image
- draw a polygon made out of smaller triangles one by one
- for each, set proper texture coordinates that correspond to it
- graphics card fills the quads with the texture

3.4.2 Personal View - choice of products

The models designed in SketchUp are used to created the Personal View. They are exported as .mesh files into a shared folder between Computer 1 and Computer 3. The meshes are loaded into OGRE and the rendering is displayed in 3D on the data panel of the LCD sandwich.

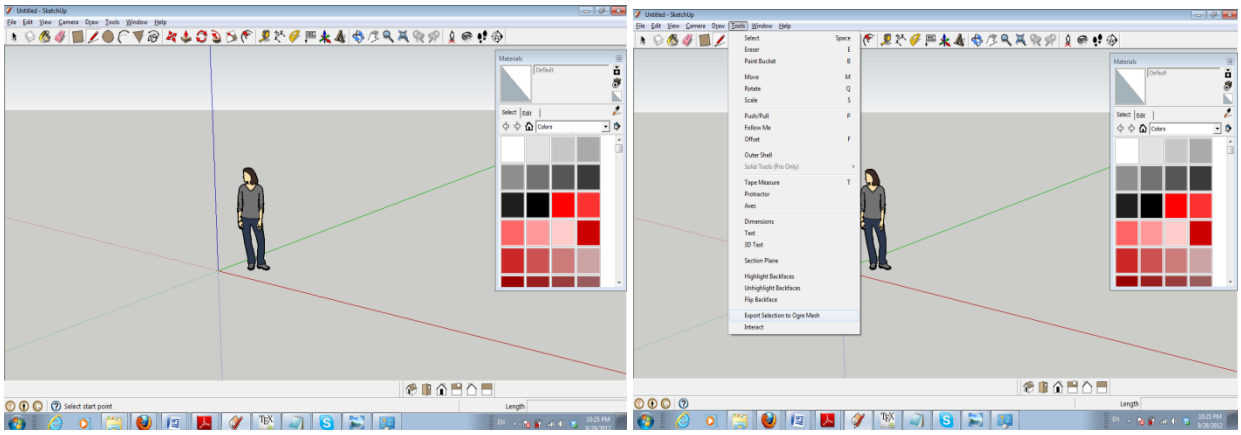


Figure 12: SketchUp supports plugins made using Ruby script – user interface for them can be created in the menu (right)

SketchUp to Ogre Exporter

In order to be able to import the models created in SketchUp to OGRE, they need to be converted to files suitable for OGRE to load and render.
For this purpose 'sketchup_ogre_export-v1_2_0beta9' is used. The exporter is a ruby based plugin which exports to an XML format. Different versions of the same plugin exists but version 1_2_0beta9 deals with exporting the meshes in a better way than the previous version of the plugin and its modifications. Such modifications are adding a diffuse component to the exported colors to simulate the models as they are colored in SketchUp. Another useful feature in the newest version is support for exporting backfaces. Version 1_2_0beta9 performs correct UV mapping export of textures and correct color export by normalising the normals of the material. An item in the Tools menu of SketchUp is used to start the export procedure.

The ogre_export_config.rb file assigns paths to variables. It sets the locations for the exported xml files and meshes, for the materials and for the textures.

beckface.rb is responsible for displaying the back faces of the surfaces in OGRE. SketchUp creates surfaces which can be visible both when looked from the front and from the back. On the other hand OGRE only displays a surface if it is looked from the direction to which the surface normal points to.

Finally, the ogre_export_1_2_0b9.rb file is responsible for recursively scanning the objects selected for export and saving their geometry in separate structures. The whole entity is converted into a geometrical object made out of triangles only. The triangles with the same material make up one submesh. All the submeshes together form the whole output mesh. These triangles are separately loaded into Ogre which means that a smaller number will result in faster mesh import.

OgreCommandLineTools

The Ogre Command Line Tools are tools that help the processing of mesh files from external to Ogre programs. When these tools are used the mesh files taken as input are overwritten by the newly output files.

- XMLConverter - Converts between the binary and XML formats for .mesh and .skeleton. Will also allow you to generate LOD information if you are converting to the binary format. This tool is necessary to convert from the XML to OGRE's native runtime format if your exporter produces XML. You can find the XML Schema for the .mesh

- Mesh Upgrader is another useful tools to upgrade the meshes exported from SketchUp to the newest mesh format used by the Ogre rendering engine. Using this application to transform the meshes is necessary for the correct loading of meshes and the mapping of the corresponding materials on them.

Loading of Meshes

A three methods for the mesh loading can be used. The first method uses one thread to create the contents of the scene, update and re-render the models. This method is the safest in terms of accessing common resources at the same time, because it uses a single thread for all the processes happening outside the Main OGRE render thread. Using it results in the same resource being accessed in different times by different processes. The drawback of the method comes from the time gap between the reloading of resource files. During this break the application thread is stopped which results in the OGRE Main thread waiting to render the created entity. The rendering is stopped and the screen 'freezes'.

The second method involves creating a second thread to reload the resource between frame renderings. Mutexes are needed to schedule when each thread becomes active. This method had its drawbacks in that it also slows down the rendering in the OGRE Main thread. This happens because the resource loading thread locks the mutex when it starts the process of reloading and releases the lock only after the new entity has been created from the updated resource. This forces the Main thread to wait until the mutex is free and then render the next frame. 'Freezing' of the screen occurs, even though it is less noticeable than the one occurring in the first method. The last method uses ResourceBackgroundQueue which is an OGRE class responsible for loading resources in the background. It registers a listener for the events of loading the meshes and receives callbacks from it once the resource groups get reloaded. At that moment it creates the new entity. The application thread method FrameRenderingQueued checks for if the mesh load is complete and replaces the old entity with the new one.

The last method proved to be most suitable for the task, because it ensures the concurrent loading of resources and mesh rendering. The mesh file and its texture load at the same time as the frame

rendering. Both threads run simultaneously without accessing the shared resource at the same time and waiting for a mutex to be released before continuing work.

To complete the final step of the design, the PiVOT API is used. Once a mesh is loaded from the shared folder its colors are transformed by the API. They appear reversed on the LCD panel. This way, when illuminated by the polarised light coming from the mask panel, the colors are re-reversed to show the original RGB values.

Threads

Where the different elements of this system come together the processes they execute need to run in one thread or concurrently in more than one threads. In the Shared View design handling the interaction with the web page needs to happen at the same time as the rendering of the web content. In the Personal View design resource loading needs to run concurrently with the rendering of meshes.

There are two choices of possible threads to use - pthreads and boost threads.
Boost has similar functionality to pthreads and is in fact a wrapper around pthreads. Most pthread concepts translated into boost functions, some are copied directly from the native functions such as the functions assert(m_Thread) and m_Thread->join() used to terminate the processes. The latter provides easy to use C++ interface in comparison to pthreads which is a C library. Creating threads with boost, and locking shared resources requires less lines of code than pthreads as it makes use of C++ syntax and coding practices. Mutexes don't need to be released, as this is automatically done as the function that has acquired the lock returns. This project makes use of several different programming and scripting languages: Ruby, C++, Java, XML - to limit the amount of different syntaxes used, boost threads are chosen to take care of the multithreading and shared resource locking.

3.4.3 Touchscreen

The interaction with both the Personal and the Shared View is designed to happen using the touchscreen. The touch gestures and commands are sent to SketchUp where instruments can be used to create, color or modify the models in some way. Only part of the touches, however, are interpreted as commands for SketchUp. Some actions are sent directly to the Ogre application responsible for producing the Personal View. The result of these actions are scaling, rotating and zooming in and out of the model. This provides an easy to interface for viewing the SketchUp model during its development. The second part of the touchscreen interface is responsible for translating the touch gestures in mouse movements in the SketchUp environment.

The interface is designed to receive all the touch gestures in one place and depending on the type of gesture make a decision which system to send the touch to - the system generating the Personal View or the one projecting the Shared View. These gestures are detected using the software that comes together with the touchscreen device itself - it is called PQ Labs MultiTouch Server. The server receives touch input from the device and can extract the coordinates of these touches. To do that, an SDK is used. The PQ Labs company provides support for a touchscreen with a C++, C Sharp or Java SDK. Windows 7 does not support the C++ SDK and for that reason the Java SDK was chosen to capture touches from the screen using Eclipse.

Using two different programming languages to create an interface creates the next design problem - how to notify the C++ Ogre applications that a touch from the user is received. The answer is - by sending a message from the Java application to the C++ application. The communication needs to happen between two programs on the same or different computers.

Two similar protocols for performing this task can be used - Midi and OSC.
OSC (Open Sound Control) is a protocol that performs hardware transport of messages between multimedia devices on the Ethernet.
The messages are made of an address and a list of one or more arguments.
[picture OSC message]

Midi is a similar protocol, but it is more music oriented. (A special cable is needed to transmit the messages from one device to another, whereas OSC uses the Ethernet connection). Both protocols can transmit integers as their parameters as binary data. OSC has an URL-style address field which makes it easier for messages to be separated and sent to different computers. Because of the address format advantage OSC messages are used to send touch commands to the Personal and Shared View systems.

3.5 System requirements

Several task need to be completed in the development of the interface for the two systems. Each task is performed by a device. One or more computers can be responsible for the tasks needed to implement a View.

A device is needed to run SketchUp and display the application window on a screen. Another task is to transfer the

The LCD sandwich and the projector need to be connected to different machines. Each machine is responsible for a different output. Computer 1 is connected to the data and mask panels and is displaying the Personal View. Computer 2 is connected to the projector and is responsible for generating the Shared View.

Each computer can only have one window in focus because of the operating system. This means that only one window can be updated at a time. As both views are actually the OGRE scene, ogre render window must be in focus in both Computer 1 and Computer 2. However, in order to capture the SketchUp viewport, it must be displayed in focus on the screen. A third computer is needed to run the SketchUp application and VNC is used to connect to this screen's desktop.

2.5.1 Advantages and Disadvantages

Using 2 computers to generate the Personal View and the Shared View and to run SketchUp is made inconvenient by the inability of a computer to have two different program windows in focus at the same time. SketchUp, the Ogre Shared View application and the Ogre Personal View application need to be the windows in focus in order to update their contents. If both SketchUp and one of the Ogre applications run on the same machine, one of them will cease to update which will result in

the other one failing to receive the updates made. For this reason, the interface requires the use of three computers. Each of these three computers will be responsible for one of the major elements connected by the interface. Computer 1 will be in charge of creating the projection for the Shared View, Computer 2 will generate the renders for the Personal View and Computer 3 will run the SketchUp application.

List of tasks completed to setup the system:

Computer 1 - 'Bob' - Installed Visual Studio 2010
- **installed OgreSDK_vc10_v1-8-1**
- **installed awesomium SDK - version 1.6.6**
- **installed PQ Multitouch Platform and Windows 7 Driver**
- **installed Java PQ Labs SDK**
- **install Java OSC**
- **include OSC pack 1_0_2**
- **include OFXMessages**
- **include boost threads**
- **set project variables include paths, library directories**

Computer 2 – 'MM' - install SketchUp
- **-install OgreTo SketchUp Exporter, modified**
- **- install OgreCommandLineTools**
- **- install ThinVNC**

Computer 3 – 'Alice' - installed Visual Studio 2010
- **installed OgreSDK_vc10_v1-8-1**
- **include OFXMessages**
- **include boost threads**
- **include OSC pack 1_0_2**
- **PiVOT API**

Chapter 4

Implementation

4.1 Setting up the Touchscreen

The touchscreen is the new piece of hardware added to the PiVOT system since its predecessor. The software for using it was added to the whole system to increase its functionality. To achieve this the PQ labs server for a Multi-Touch G^2 Touch Screen was installed on the MM computer. The PQ Labs Multi-Touch SDK is installed on the same computer and Eclipse is configured to work with it. The new project properties are changed to add the Java OSC libraries and PQ libraries:

lib/dom4j-1.6.1.jar
lib/PQMTClientJava.jar
lib/PQOSUtil.dll
lib/x64/PQOSUtil.dll

javaosc.jar
javaoscfull.jar
junit.jar

The events from the touchscreen are captured using a window created by JFrame - a framed window of size 1024x768 pixels. A WindowAdapter is initialised and used to receive window events. A frame listener is registered. Before it starts listening for touch events the connection to the server needs to be established and the client request type. The client request type defines which

requests from the client are being recognised - in this case all raw the raw data and gestures are recognised by the listener. Finally, the resolution of the multi-touch server it taken.

To capture the events from the touchscreen
the PQMTClient class is extended and some of its methods are overriden. One such method is 'OnTouchGesture'. The method listens for touch events of different types, creates and OSC message with the parameters of this action and sends this message to corresponding computer. The actions that this application listens for are of types shown in the table.

The address of the event is given and the arguments can be the x and y coordinates of the touch, the relative difference between the previous and current positions of the touch before and after the move.

TG_CLICK	6669	/PQLabs/MClick	MM
TG_DOWN	6669	/PQLabs/MDown	MM
TG_UP	6669	/PQLabs/MUp	MM
TG_MOVE	6669	/PQLabs/MMove	MM
TG_SPLIT_APART	6668	/PQLabs/SApart	Alice
TG_SPLIT_CLOSE	6668	/PQLabs/SClose	Alice
TG_DB_CLICK	6669	/PQLabs/MDClick	MM
TG_MULTI_MOVE_UP	6668	/PQLabs/MultiU	Alice
TG_MULTI_MOVE_DOWN	6668	/PQLabs/MultiD	Alice
TG_MULTI_MOVE_LEFT	6668	/PQLabs/MultiL	Alice
TG_MULTI_MOVE_RIGHT	6668	/PQLabs/MultiR	Alice
TG_ROTATE_CLOCK	6668	/PQLabs/Clock	Alice

TG_ROTATE_ANTICLOCK	6668	/PQLabs/Anticlock	Alice

The table shows the implemented touches along with the OSC message parameters that are sent. The Address column displays the receiver of the message and Arguments column shows the parameters of the touch. The information in these two columns makes up the OSC message. InetAddress is a class which consists of the IP address of a device - a 32 bit or 128 bit unsigned number. Two such objects are created and used to initialise two instances of the class OSCPortOut. The class consists of an IP address to send the messages to and a port number to use. The two instances contain the addresses of Alice and MM. Depending on the event type received from the PQ Labs server, the OSC message is sent to the corresponding device.

The method for sending OSC messages to another computer simply checks if the built in the OSCPortOut class command 'send()' can be called and throws an exception if the port with the given number is in use or of the IP address is incorrect.

4.2 Implementing the Shared View

The process of achieving a Shared View on the display consists of creating a link between the following software products and processes:
- ThinVNC - capture the SketchUp viewport and inject it in a browser
- Awesomium library working on Ogre - import the capture in the renderer
- calibration - Ogre, build polygons, create textures, map them
- Ogre processing- reshape/resize the image according to the pre-calibration
- boost threads to receive OSC messages from the touchscreen

4.2.1 Capture SketchUp Viewport with VNC

The aim of this first step is to link the computer that will be connected to the projector to the computer running SketchUp using remote screen sharing. VNC allows screen sharing, file transfer and remote desktop connection. SketchUp is installed on computer called 'Bob'. ThinVNC is installed on the same computer and an IP address and port number to bind to are taken note of. This address and port are used by other computers on the same network to establish a connection and start the screen sharing service. Bob runs the SketchUp application and leaves its window in focus. The screen resolution of the monitor is 1024x768.

Computer 2 - 'MM', opens a web browser and connects to Bob's address using the port number in the way shown in picture 10. The URL is a concatenation of the IP and port number separated by a colon. MM sets its screen resolution to match Bob's - 1024x768.

The service starts when the 'Connect' button is pressed. The desktop of Bob is then shown on MM

displaying the SketchUp viewport.

4.2.2 Render the Web Browser in OGRE

After the connection between the two computers through VNC is successfully tested, the contents of the web browser need to be input in OGRE using the Awesomium library. For this purpose OGRE SDK is installed on MM together with Microsoft Visual C++ 2010 Express and the Awesomium library. To set up the new application the project configuration properties are modified. The Ogre and Awesomium include directory locations are set and the library locations are given as additional library directories. The .lib files that need to be included are explicitly specified in the additional dependencies section of the linker property page in the project configuration window.

The coordinates of the vertices calculated during the calibration are input as a text file. They are read one line after the other and the values along the X and Y axis are put in two arrays. The first array contains the X coordinates and the second - the Y coordinates. These arrays are public and can be read by the function that takes care of creating the polygon plane and assigning texture coordinates to the vertices of the each polygon. This function is described in more detail in the Calibration section in this chapter.

The projected image is shown on the screen in the corresponding shape. ThinVNC is used to capture the viewport. It captures the SketchUp screen and loads it into a browser. The browser content is then rendered in OGRE. The image is calibrated by mapping it to a polygon shape in OGRE with precomputed coordinates. The resulting image is sent to the projector.

The structure used to embed the web view into OGRE is called class SketchUp. It consists of a web view structure taken from the awesomium library. Awe_webview is the awesomium equivalent of a tab in the web browser. This tab can load a URL, refresh the content, interact with the this content and finally render it. The Ogre::TexturePtr gives a pointer to the texture that will correspond to the same webView.

```
class Sketchup{
public:
awe_webview* webView;
Ogre::TexturePtr texture;

};
```

The webcore of Awesomium is responsible for the webView. A new instance of it is created and called to create the webView that will be used to display the screen sharing service of VNC. The size of this web view is defined when creating it as 1024x768.

```
awe_webcore_initialize_default();
newWin->webView = awe_webcore_create_webview(1024,
                                768,
```

```
                    false);
```

The URL string VNC will connect to is input by reading a line from a text file. The URL is formed in the same way as described in the last section - '172.21.207.180:8080'. This string is passed to a function in the awesomium library and loaded in the WebView..

```
awe_string* url_str = awe_string_create_from_ascii(mon.c_str(),
  mon.length());

awe_webview_load_url(newWin->webView,
            url_str,
            awe_string_empty(),
            awe_string_empty(),
            awe_string_empty());
 awe_webview_focus(newWin->webView);
 awe_string_destroy(url_str);
```

A new Ogre material is created with the name 'SketchupMaterial' and a texture called 'SketchupTexture' is assigned as a Pass to it. This new texture becomes the texture used in the instance of class SketchUp used in the application. A library function is used that waits until the web page is loaded completely into the webview and then updates the render buffer of this webView.

```
Ogre::TexturePtr tex = Ogre::TextureManager::getSingleton().createManual(
             "SketchupTexture",
            Ogre::ResourceGroupManager::

                                        DEFAULT_RESOURCE_GROUP_NAME,
             Ogre::TextureType::TEX_TYPE_2D, 1024, 768, 0,
             Ogre::PixelFormat::PF_BYTE_RGBA,

                                        Ogre::TU_DYNAMIC);
```

HBU_DYNAMIC	Modifying the contents of this buffer will involve a performance hit. Indicates the application would like to modify this buffer with the CPU fairly often.

```
newWin->texture = tex;
while(awe_webview_is_loading_page(newWin->webView))
        updateCore();
```

This texture is assigned to an entity, which in this case is a polygon, and the entity is attached to node in the Ogre scene. During the Main Ogre thread the entity is rendered.

4.2.3 Update the Contents of the Web Page

In the previous section it was shown how to render a web page from a given URL. This section explains in detail how the entity along with its texture is updated as the content of the web page changes.

The function to update the contents of the WebPage is called between frame renderings in the FrameRenderingQueued function. It waits for the new content of the page to fully load before updating the webcore in the same way as described in the previous section.

To update the web page a render buffer is created. It is a 32 bit pixel buffer of raw pixel data of the WebVeiw in BGRA format (Blue, Green, Red and Alpha channels; Alpha channel defines the level of opacity). "Renders this_WebView into an offscreen render buffer and clears the dirty state. Returns a pointer to the internal render buffer that was used to render this_WebView. This value may change between renders and may return NULL if the_WebView has crashed."

"An array of unsigned chars is set to point to the same buffer. A raw block of pixel data, BGRA format is also created using the awesomium function. 'Pixels' is a pointer to a 32-bit BGRA pixel buffer."

```
const awe_renderbuffer* renderBuffer = awe_webview_render(newWin->webView);
const unsigned char* pixels = awe_renderbuffer_get_buffer(renderBuffer);
```

In the initialisation step described in the previous section it was shown how the texture of the SketchUp class instance is set to point to the SketchupMaterial. Here, the pointer to this texture is used to create a pointer that will be used to share the pixel buffer. The pixel buffer of the Sketchup texture is made to point to the same memory location as the newly created shared buffer. This buffer is used to copy and change data modifying the texture itself.

```
Ogre::HardwarePixelBufferSharedPtr pixelBuffer = newWin->texture->getBuffer();
```

The hardware buffer is locked in order to update it. This step involves creating a pointer to work with and informs the underlying hardware card that an access to the buffer written on it is about to happen. Different lock options can be set to the process. In this case, the lock is set to discard the the whole contents of the buffer. After the buffer is modified, it is unlocked.

```
if(renderBuffer != NULL)
  pixelBuffer->lock(Ogre::HardwareBuffer::LockOptions::HBL_DISCARD);
 Ogre::PixelBox pixelBox = pixelBuffer->getCurrentLock();
```

A PixelBox is a primitive structure containing an image in pixels in memory, a 2D array of pixel values. A new object of this class is created and the locked region from the buffer is assigned to it. With the lock option being HBL_DISCARD this region is the whole buffer. When the pixel values of the PixelBox are modified, this results in creating a new pixel buffer in the card memory which modifies the texture this memory location points to.

```
Ogre::uint8* pDest = static_cast<Ogre::uint8*>(pixelBox.data);

for (size_t j = 0; j < 1024; j++)
  for(size_t i = 0; i < 768; i++)
  {
    *pDest++ = pixels[((j*768)*4)+(i*4)+0]; // B
    *pDest++ = pixels[((j*768)*4)+(i*4)+1]; // G
    *pDest++ = pixels[((j*768)*4)+(i*4)+2]; // R
    *pDest++ = pixels[((j*768)*4)+(i*4)+3]; // A
  }
```

The raw awesomium pixel buffer is used to assign the new values to the Ogre RenderBuffer. The Blue, Green, Red and Alpha values of the array of the 1024x768 unsigned chars is assigned to the destination buffer of 8-bit unsigned integers pointing to the same memory location as the pixelBox. This destination buffer changes the pixelBox data thus assigning new values to the Render Buffer in memory and modifying the texture. This texture was assigned to an entity in the initialisation step and during the Main Ogre thread the entity is rendered with the updated texture.

4.2.4 Inject Events in Browser

This step aims to organise and enable the receiving of OSC messages with user commands from the touchscreen. This is achieved by registering a packet listener. This listener is binded to a socket using the IP address of the sender and port number of the communication. This socket is the same socket the Java application running the PQLabs SDK is using to send its OSC messages. The whole communication between the two applications goes through this port. The listener runs on a thread concurrently to the main application thread and the Ogre Main thread until there is a signal from it receives a signal to break. The break in this case is called when the application terminates.

The listener is bound to port 6668 and receives callbacks whenever an OSC message is received from the UDP channel. A queue to store the OSC messages is created as a shared resource.Every time it receives a message it is processed in the following way:

Upon receiving the OSC message, this message is converted to an ofxOscMessage, by copying the OSC address to the new message, and also the unsigned long value host and port number. The queue is locked and the message is pushed at the back of the queue. Finally, the queue is unlocked. This function makes use of the ofxOscMessage function calls to put the incomming message in the

queue. A mutex is initialised to lock and unlock this common resource which is accessed from several methods.

```
        boost::mutex oscMutex;
    std::deque< ofxOscMessage* > messages;
```

Between rendering every next frame in the Main Ogre thread several processes take place. The queue is checked for any waiting messages by checking weather its length is bigger than 0. If messages are waiting the first one is popped out of the queue. This address of the OSC message is then checked against several cases and the corresponding events are injected into the browser at the exact coordinates received in the message. During all these operations the queue is locked with a mutex to prevent the simultaneous read and write of memory.

```
if ( m.getAddress() == "/PQLabs/MMove" )
  InjectEvent(PQ_M_MOVE, m.getArgAsInt32( 0 ), m.getArgAsInt32( 1 ), 0);
```

```
inject event
switch(injection)  {
  case PQ_M_MOVE :
     awe_webview_inject_mouse_move(newWin->webView, mx, my);
        awe_webview_inject_mouse_up(newWin->webView, AWE_MB_LEFT);
     awe_webview_inject_mouse_down(newWin->webView, AWE_MB_LEFT);
```

Injecting mouse presses and moves requires the arguments current mouse coordinate x and current mouse coordinate y. The coordinate system on the touchscreen mirrors the OGRE coordinate system ranges between 0 to 1024 for the x coordinate and 0 to 768 for the y coordinate. The values are input as arguments to the awesomium command injection operations. Corresponding actions are triggered when the application receives OSC messages from the touchscreen.

PQ_M_MOVE	awe_webview_inject_mouse_move(newWin->webView, mx, my);
PQ_M_DOWN	awe_webview_inject_mouse_move(newWin->webView, mx, my); awe_webview_inject_mouse_down(newWin->webView, AWE_MB_LEFT);
PQ_M_UP	awe_webview_inject_mouse_up(newWin->webView, AWE_MB_LEFT);

PQ_CLICK	awe_webview_inject_mouse_move(newWin->webView, mx, my);
	awe_webview_inject_mouse_down(newWin->webView, AWE_MB_LEFT);
	awe_webview_inject_mouse_up(newWin->webView, AWE_MB_LEFT);

When running the application the result is the rendered web view of VNC connected to Bob. As mentioned earlier, the service starts when the 'Connect' button is pressed. In order to automatically force this behaviour a command in injected to the web browser using awesomium. This command is a simple button press at the pixel coordinates where the button is located. The event is triggered and the application starts the service, capturing the screen of the other computer thus rendering the SketchUp viewport.

```
listener.InjectEvent(PQ_CLICK, 324, 294, 0);
```

4.3 Implementing the Personal View

Steps to implement the Personal View are based on establishing the connection between the software and creating a pipeline for sharing the resources

- Ogre to SketchUp exporter - modify exporter to automate mesh conversion
- create a shared folder between two computers
- Ogre - import meshes from shared folder
- Pthreads, boost threads - reload meshes, materials, textures
- PiVOT api - display models in 3D
- OSC messages - receive commands from the touchscreen

To implement the Personal View interface, an open source exporter is modified. The modifications include automating the export.

4.3.1 Ogre to SketchUp export

The first step in the implementation of this interface is automating the way models are exported from SketchUp. For this purpose 'sketchup_ogre_export-v1_2_0beta9' is modified. The main function in the script is changed to export continuously every 5 seconds. A clock is added to determine when the next export will take place. All the user notifications are blocked and the OgreXMLConverted is made to run in minimised mode in order for the command prompt to become less visible when exporting the meshes.

A shared folder is created between 'Bob' - the computer running sketchup, and Alice - the computer responsible for generating the personal view. This shared folder is created on Bob. A network is created between the two computers, so Alice can access Bob's shared folder. The network that is set up using a crossover cable is a home network with enabled file sharing. All computers in this

network are given the permission to access the Shared Folder on Bob. All the meshed in the SketchUp environment are exported under the name box.mesh, box.mesh.xml and box.material to the shared folder. All the textures are exported to a subfolder of the shared folder.

4.3.2 Importing the Models in Ogre

This step deals with importing the models which the SketchUp exported to the Shared Folder. To start with Ogre needs to be configured to load the models from the shared folder. For this purpose a new resource group called 'Shared' is created and the location of this shared folder is set as its input directory. The first design can be implemented fairly easily by simply creating an application thread to run concurrently with the Ogre Main thread and reload the updated model called Box from the Shared resource group. The algorithm is as follows:

Main Ogre thread: RenderScene();

Application Thread:
receive a callback after the current frame is rendered

check for any new OSC messages and execute the touchscreen command
sleep(MS)
destroy the resource group Shared
initialise the resource group Shared
detach old entity from scene node
destroy old entity Box
create new entity Box
attach Box to the scene node

This algorithm is convenient in the sense that the shared resources are accessed by two threads at different times and therefore it is thread safe. It's drawback, however, is that the Main Ogre thread has to wait until all of the steps in Algorithm 1 are completed before rendering the next frame. The model update is scheduled to happen once every 3 seconds, which means that the Main thread has to wait for the application thread for at least 3 seconds before re-rendering the scene. This method is straightforward by the stalls make it not suitable for generating the shared view.

Algorithm 2 achieves much better results. It has the following steps:

Main Ogre thread: RenderScene();

Application Thread:
unlock mutex
receive a callback after the current frame is rendered

check for any new OSC messages and execute the touchscreen command
destroy old entity Box
swap pointers of Box with BoxNew
attach Box to the scene node

lock mutex

Resource Loading Thread
sleep(MS)
lock mutex
destroy the resource group Shared
initialise the resource group Shared
detach old entity from scene node
create new entity BoxNew
unlock mutex

This algorithm performs a mutex lock at the end of the Application Thread which runs between frame renderings. The resource loading thread will need to wait until the frame is rendered to acquire the lock. This method ensures that the resource and entity are destroyed at a time when the Main Thread will not access them. With this method the rendering thread does not have to wait during the 3 second gap between reloads. However, it still has to wait for the Application Thread to acquire the lock. To get the lock the Application thread has to wait for the Resource Loading Thread to release it which creates stalls in the rendering of frames.

Algorithm 3 - ResourceBackgroundQueued
The third algorithm makes use of a thread which loads resources in the background. It accesses the resource concurrently with the Main Ogre thread. It separates the tasks of resource reloading into reloading the resource group and reloading the entity itself. The resource loading thread continuously checks what part of the resource reloading is complete. If the resource group has not completed parsing the texture scripts or reloading the meshes, it does not wait forcing the Rendering wait, but exits and performs another check on the next iteration. Once the resource group has finished reloading the new entity is created. Again the
main thread does not wait for this entity to be created but renders the non-updated one. At one point (in between the frames) the background thread receives a callback to resume the model load. It creates a new entity called BoxNew and notifies the Application thread that the new entity is ready. Identifiers are initialised and they denote the the readiness of each resource loading step. The Operation Completed function is a callback function of the background resource loading thread alerting the application that a background loading operation was completed.

Algorithm 3

Main Ogre thread: RenderScene();

Application Thread:
receive a callback after the current frame is rendered
check for any new OSC messages and execute the touchscreen command
checks if the updated entity is created in the background ⇔ check = 1
destroy old entity Box
swap pointers of Box with BoxNew
attach Box to the scene node

sets check back to 0

Resource Loading Thread
sleep(MS)
clear the resource group Shared
initialise the resource group Shared in the background

OperationCompleted
receives identifier from the background thread
checks if this identifier matches the one denoting finished resource group initialisation
starts the loading of the .mesh file

else if identifier matches identifier for finished .mesh loading
new entity is created from the loaded resource box.mesh
public variable check is set to 1

This method makes sure that the shared resource are not accessed at the same time. If part of the resource loading process in not complete, the rendering thread does not wait, but continues rendering until the updated entity is created. Then, the Application thread does the swap and the updated model is rendered on the next iteration of the Main Ogre thread.

4.3.3 Interacting with the Personal View

The communication with OSC messages between the touchscreen and the Ogre application is implemented in the same way as in the Ogre application generating the projection. The messages received in this case are caused by actions performed with multiple fingers by the user. He or she is able to scale, rotate and translate the model in order to view it from different direction and of different size. The action of moving multiple fingers up translates the model up by 3 units in the Ogre coordinate system. A corresponding action occurs if the user of the touchscreen moves multiple fingers down, left or right. If the user performs a rotating movement with his fingers in a clockwise direction the model will rotate clockwise along its y axis. Finger rotation in the opposite direction will result in an anticlockwise rotation of the model. Finally, splitting two fingers apart on the screen will zoom in the the model and closing one's fingers results in zooming out of the model. These touches along with the corresponding changes in the Personal View are visualised in the Appendix.

4.3.4 Displaying the 3D Models

This step involves integrating the application for loading meshed from a shared folder and rendering them into the PiVOT 3D mesh display system using the PiVOT API. The scene node that has the canvas attached to it is rotated and translated in a way that mirrors the view of the SketchUp viewport. The OSC library was implemented in the PiVOT API to allow the communication of commands between the touchscreen and PiVOT system displaying the models in 3D on the LCD in the Personal View.

4.4 Calibration

The process of calibrating the image that will be projected on the screen is performed once. Its main purpose is to create the dimensions of the entity that the SketchupTexture reflecting the web browser running VNC will acquire. This entity will be resemble the shape of a canvas sail of a ship. When the web content gets updated, the texture of the entity changes, but its original configuration

and setup remain.

To setup the vertices of this 'canvas' and map a texture to it, the procedure is as follows:
- start with a square shape, made out of 4x4 squares. Each square is made out of 2 triangular polygons connected at their longest side.
- Separate the texture into 4x4 squares and map its vertices to the canvas vertices.
- Move each of the vertices manually until it reaches the correct position. Update all the polygons and textures. Repeat this step until the correct shape of the canvas is achieved.
- The resulting projection of the canvas should be a rectangular image.
- Once the vertices arrangement is final save the x and y coordinates to a text file

Setup Procedure

Step 1: The coordinates of 25 points chosen in a way that will arrange them in a grid are put in a two dimensional integer array. The points will be used to create the 2x16 = 32 triangular lists that will make up the 16 squares of the canvas.
Step 2: A triangular list is created using the (x, y) coordinates of 3 points in the two vertex array.
Step 3: Three sets of (u,v) coordinates of the corresponding points of the texture are mapped to the vertices of the triangular list. These coordinates range from 0 to 1 in both u and v directions.
Step 4: The vertices of the polygon are build in an anticlockwise direction in order to be visible in Ogre.
Step 5: The triangular list is attached to the Scene node
Step 6: Go back to step 2 and use the next set of three (x, y) vertex coordinates and three (u,v) texture coordinates

Update Procedure

Step 1: Select the current node to be modified and save its ID as a public variable
Step 2: Listen for a keypress from the user.
Step 3: Using the arrow keys the user can modify the values of the x and y coordinates with a step of 3 units in the Ogre coordinate system. The values of the IDth item in the two dimensional array of vertices are also updated.
Step 4: Get the 32 polygons one by one and update their vertex locations. The mapping of the (u, v) coordinates of the texture remain the same.
Step 5: Go back to step 1 and repeat the procedure until the correct canvas coordinates are achieved. Save the resulting vertex x and y values to a text file.

The text file is loaded at the beginning of the projection application and the polygons along with their corresponding texture mappings are generated. The resulting 'canvas' is the collection of entities attached to the Ogre scene node. Each of the entities displays a piece of the whole texture, which updated will display the updating browser. All the pieces together act as one and display the updating SketchUp viewport captured by VNC. This is also the required shape for the projector to display.

```
man = mSceneMgr->getManualObject("VertexB" +  Ogre::StringConverter::toString(id));
man->beginUpdate(0);
man->position(vertexX[id], vertexY[id], 0.0);           man->textureCoord(uvX,uvY-0.25);
man->position( vertexX[id+1], vertexY[id+1], 0.0);     man->textureCoord(uvX-0.25,uvY-0.25);
man->position( vertexX[id+5],  vertexY[id+5], 0.0);    man->textureCoord(uvX,uvY);
  man->index(0);
```

```
  man->index(1);
  man->index(2);
  man->index(0);
man->end();
```

4.5 Improving the Performance of the Interface Components

Keep Ogre and SketchUp windows in focus
The touchscreen interaction reflects both on the projection screen and the computer screen. This can cause Ogre to lose focus and become an inactive window. Both Ogre and SketchUp need to be in focus in order to update their contents. To force this the software Actual tools is used to make window remain in focus all the time.

Texture Handling
A second system call was added to the plugin script which deletes the old textures from the Shared folder before generating the new ones. This is necessary due to the fact that OGRE reloads the whole texture resource group and parses the scripts in this group. If the model changes and some textures are exchanged with other textures, the unused resource files need to be deleted from the resource group to decrease the loading and parsing time of the renderer. Another way of dealing with this problem is loading all the textures into Ogre at the start of the application and reload only mesh files and materials later during the background resource loading. This speeds up the model rendering, but has a drawback - new textures can not be added during the application execution.

Consume less CPU power
instead of using the Awesomium built in update function between frame renderings an update function is created. This function sleeps for a bit before updating the web core. This small fix eases the work of the CPU while the web page is still loading.

Export of meshes
Occasionally the resource loading coincides with the model export. In order to avoid accessing the same resource in the Shared Folder at the same time a small lock file is created before SketchUp is about to save the updated mesh and material. After the export is completed, the lock file, which is a text file of size 1K, is deleted. On the other hand, the Ogre application checks the existence of the same file in the folder. If the file exists the Ogre process continues and makes another check on the next iteration. If the file does not exist, it is created after which the resource reloading takes place. In the end of the reload, the file is deleted and SketchUp can export the updated models to the Shared Folder.

Improving touch event listening quality
The properties of the PQ Labs sever were changed to increase the area of the detected touch and become less sensitive to touches. This is necessary in order to detect the touch of the SketchUp user in an adequate way and be able to correctly select the drawing instrument. If the area of the touch is too small the touchscreen can recognise it as more than one touch and if the sensitivity is too high the screen will detect an event even if the user has only hovered his finger close above the surface. Saving the last mouse state is used to determine what the last action of the user was. If the this action was touch press down then another event TG_DOWN can not be triggered until the mouse state is changed back to TG_UP. This technique decreases further the noise coming from the touchscreen.

Chapter 5

Evaluation

5.1 User Evaluation

A group of 10 participants took part in an experiment designed to test the performance of the interface. They we asked to perform a series of simple tasks on the touchscreen and then evaluate their experience in a survey. The tasks involved working on Google SketchUp to create models and interact with these models both on the Shared and Personal Views.
After each of the tasks described above the users were asked to fill the survey. The questions were taken from the NASA TLX, which is a tool used to evaluate the workload on the participants working on the display. This workload is measured on the base of six ratings procedures. They assess the mental, physical and temporal demands of the task, self evaluation of the performance and the effort and frustration in the course of the experiment. All these scales are used to evaluate the user's workload and the user's experience of the interface. This experience is one of the two main components used to form an overall evaluation of the performance of the system.

5.1.1 Experiments

The workload in this environment is mainly defined by the response of the system to touches on the tabletop. That's' why the 10 participants (aged 22 to 30) were asked to complete 3 tasks on the system and mark their experience. Each participant was given information about the system and the way its two views can be observed. Each participant was then sat in a chair in front of PiVOT

and some time to get used to switching from one view to the other. Some participants were leaning forward above the screen to see the Personal View and others were standing up.

Tasks:

Three simple tasks were created to measure the experience of the users working on the display system. They involve using the touchscreen to create and move objects shown on the PiVOT screen. Firstly, the users were given a trial session on the system in which they had to move shapes drawn on SketchUp. The aim of the trial session is to introduce the participant to the PiVOT display and to the touchscreen interaction.

Task 1

In this task users were asked to rearrange the furniture in a house drawn in SketchUp. The models of the furniture were placed in the wrong rooms before the task starts (for example the bed was placed in the living room). The aim of the participants was to use the touchscreen to put the objects in their corresponding rooms. The task evaluates the workload of the participants switching between the Personal and the Shared Views while performing simple operations. The users had to move the following items – bed, bench, kitchen table, living room table, sofa and toilet, to one of the following rooms – living room, bedroom, kitchen, toilet and closet.

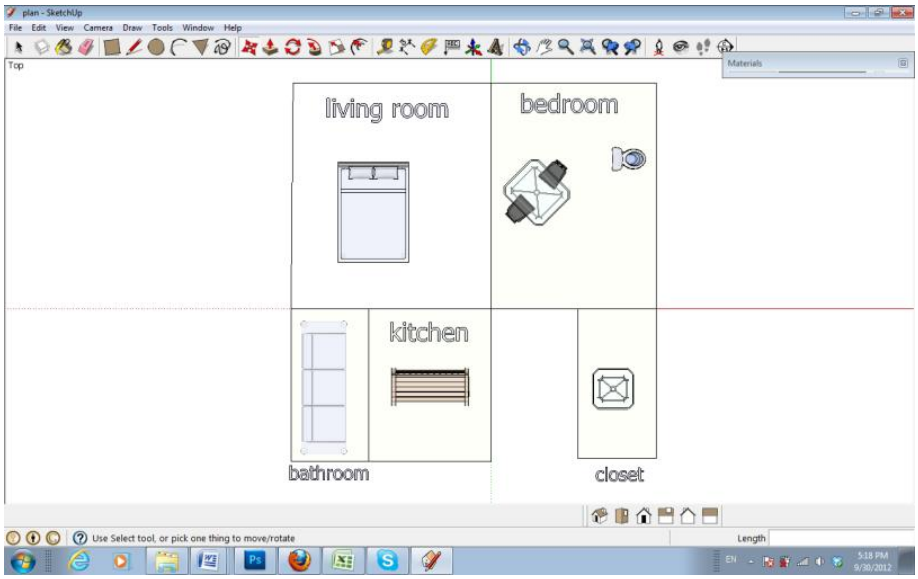


Figure 13: Task 1- Shared View

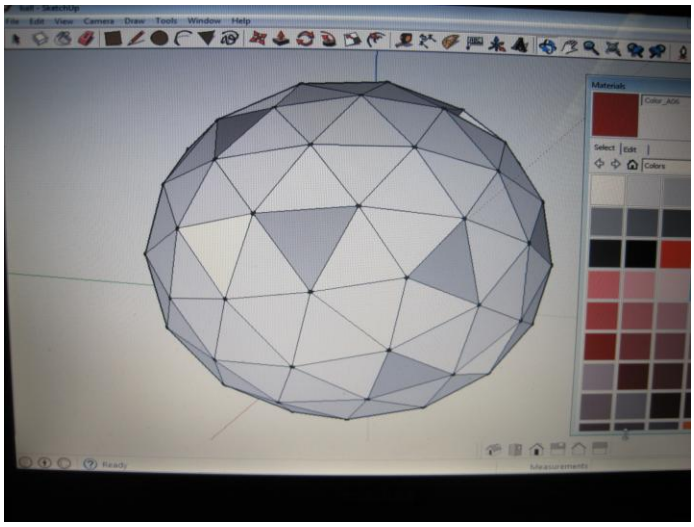
The objects are seen from the camera situated parallel to the (x, y) plane. To complete the task only one instrument from the SketchUp tools is used – the Move tool.

- The participants were not told what every object is.
- They were only told that the second view of the system shows the rendered models. They were constantly switching between the Personal and the Shared view to find out what every picture they saw in the SketchUp viewport represents.
- The participants were not told how to use the tools in SketchUp; they used their own intuition to pick, move and drop objects using the touchscreen.

Task 2

In the second task, the participants were asked to follow series of instructions to create objects in SketchUp. They had to draw the objects listed below.

- draw a square (square tool)
- extrude it to a cube (extrusion tool)
- color the different faces in different colors (paint tool)
- draw a circle
- extrude it to a cylinder
- color it



Task 3

In this task the users were shown the model of a ball and asked to find the hole in this ball. They were given access to the 3D rendered mesh and to the model on SketchUp. They were allowed to orbit around the ball in the Shared View and rotate the rendered mesh in the Personal View.

Figure 14: Task 3- Shared View

5.1.2 Experimental Results

Success Rate

Task 1

All participants managed to complete the task. All of them used the Personal View before achieving the correct arrangement. Eight out of ten people placed the bench outside after having looked at the Personal view, which is also the correct place for it.

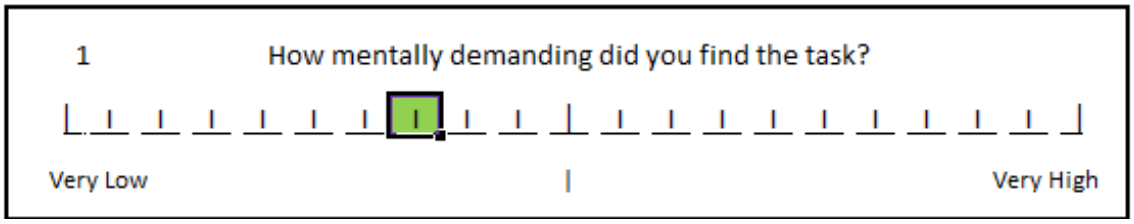
Task 2

All participants were able to draw the objects listed above. It can be seen from the table of Average Times below that the average time it took the participants to model the cube is almost twice the average time it took to model the cylinder. Every participant took more time to draw the cube, than the cylinder

Task 3

Five of the participants managed to complete the experiment. All of them managed to find one of the holes in the ball but some could not locate it in the Shared View.

5.1.3 Questionnaire Results and Discussion



The values in the table show the average of all the answers the participants gave to this question.

	task 1	task 2	task 3
Question 1	-3.7	-3.3	-1.7
Question 2	-2.7	-2.2	-2
Question 3	-3.8	-3.2	-3.9
Question 4	3.9	3.9	0
Question 5	-1.2	-1	-2
Question 6	-0.9	-2.9	-2.2

Table – task 1 – 'Furniture arrangement' – use of both Personal and Shared Views
Task 2 – 'Graphics Creation' – use of Shared View only
Task 3 - 'Hole in the Ball' – use of both Personal and Shared Views

The average scores between the participants clearly show that participants found the third task most mentally demanding and the first task least demanding. The workload in terms of mental demand does not show to be dependent on the addition of a second view. As expected that the task which asked participants to find something was the most menatally demanding. According to the

table of times needed to complete the different tasks, task 3 took the longest time to complete – 5.1842 seconds on average, which shows it was harder than the other two tasks. The difficulty in the task comes from not being able to find a hole in the ball and spending a long time looking for one, which is a mental challenge.

The second series of scores do not differ from each other a lot. This shows that participants found the tasks almost equally physically challenging. However, the third task again shows a slightly higher rating than the other two. This is a result of the participants continuously rotating the ball in the Personal View to find a hole and using the orbit tool in SketchUp to get to the corresponding location of the ball in SketchUp. Task three is the only one marked on avarage by participants as more mentally than physically demanding. However, the difference between the mental demand of Task 1 and 2 with Task 2 is not bigger than the difference between the physical demand between Task 1 and 2 with Task 3. Moreover, all the ratins are in the lower half of the grid used in the questionaire. It can be concluded, that the physical workload of using the touchscreen is not great, even for more mentally demanding tasks.

The survey shows that the participants were most confident about their success in completing the first and second tasks. According to the results, on average they worked slightly less hard to achieve that success rate. The success rate is correlated to how hard the participans worked to achieve the result - the paticipants showed they worked hardest to complete the second task. This is due to the fact that they had to follow the written directions and create objects instead of viewing and moving objects as in the other tasks. During the first part of the second task participants got used to using the interface and were able to achieve similar results a lot quicker when they have to create another object.

The participants on average gave a correct assessment of their own success rate in Task 3. They also marked this as the least rushed and hurried taks out of the three. On the other hand they incorrecly assessed the success rate of Task 3. The participants also found this task to be most rushed. They we only using the Shared View and did not see their models after they are rendered. This shows that the users need to have different views of what they are doing especially when working with more different instuments to create 3D grahpics. They found the task most rushed when in fact it took them more time to comple it that the first task, where they had a second view of the model and were more aware of how long and how hard they have to work to complete the task.

Task 3	Task 2 -part 1	Task 2 – part 2	Task 2	Task 3	Average
3.78571	1.777142857	2.92	4.69714	5.184285714	Time

Table : Average times

Key Points:

- No correlation between mental and physical workload established. This means that the users were not finding the use of the interface more difficult for harder tasks.

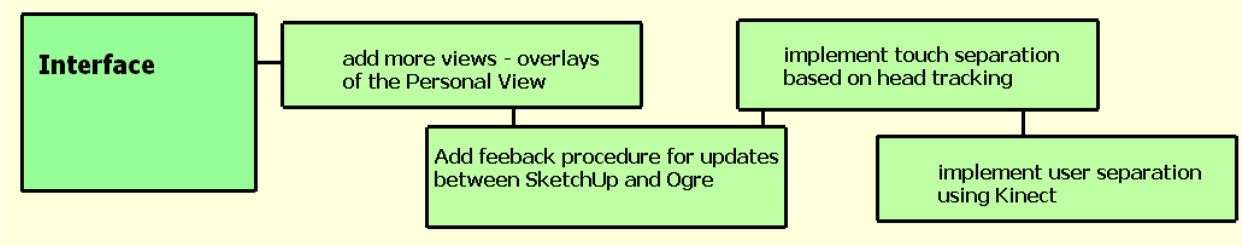
- After completing a certain task participants were able to complete a similar task nearly twice faster. This shows that working on the touchscreen is intuitive

- The interface between the two views is adequate, made participants feel less rushed and in this way gave them a more realistic idea of their own success in completing a task.

Chapter 6

Future Work

6.1 Building Up on the Interface to Support Other Real-time Graphics Applications



The interface developed for the PiVOT system can be extended in several directions to increase its functionality and application as shown in Figure 4. Building on the interface includes changing the interaction between the users of the touchscreen and the system and the generated views, introducing head tracking techniques and enhancing the overall performance and speed.

Personal View	Shared View
users can read and modify the 3D model using one-finger touches	users can interact with the 3D model without modifying it using multiple-finger touches

6.1.1 Add more views – Personal Overlays

The information shown on the two views doesn't need to be related in a special way. The Personal View can be further developed to make use of the mask panel. The result of that will be creating different location dependent Personal Views for different users. This can be achieved by

implementing the Personal View Overlays into the application. Further camera calibration and trackers for each user will be needed. To extend the view into several smaller Personal View Overlays will create a personal space on the screen for each user, where they can view some specific information about the model. For example different overlays can show the model in different coloring, shading and placed in differently lit scene, or interacting with one or more other models.

This can further extended into creating stereoscopic views of the models. The user or users wearing head tracking devices will be able to see the objects in 3D. This can be achieved by further development in the area of dynamic mask generation and calibration of the camera to support more than one user. The camera settings need to be adjusted and the PiVOT API upgraded to be able to receive and process input from the optical tracking device. Users will be distinguished by the tracker they are wearing. The touches on the tabletop can also be distinguished. Touch made by user 1 should cause a certain user 1 command to be sent. The touch made by user 2 at the same time will be distinguished as user 2 commands and the corresponding modeling events will take place. The view of a user can be private – only he will be able to see the information displayed. At the same time this same user will not be able to see the other personal views. This can be applied to giving some users special authority. For example only a specific animator can see a certain part of the model in development without any of the other co workers aware of what this view shows.

Personal View	Shared View
Updates made in SketchUp are reflected in the render Users can modify the 3D model	Updates made in any view zone are reflected in both views Users can modify the 3D model

6.1.2 Update Feedback - Editing the Models from Both Views

Currently, the Shared View has both 'read' and 'write' access and the Personal View has only read access. To add more functionality to this display system, the interface can be modified to allow write access of the model from both Personal and Shared Views. This can be accomplished by feeding information back and forth between the two views. SketchUp will update the mesh once the user modifies the model in the Shared View. If this user modifies the model in the Personal View, Ogre will need to send update information back to SketchUp so that the objects displayed in the two views correspond to each other. Sending information to SketchUp can be achieved again by using OSC messaging. Commands will be passed from the Personal View application to the Shared View application which will forward them to SketchUp through the screen sharing service running. These messages, when passed to the SketchUp viewport will execute ruby scripts that will update the SketchUp model depending on the changes made in the mesh in Ogre.

Personal View	Shared View
users can interact with the model using multi-touch	users can interact with the model using multi-touch

6.1.3 Separate touches depending on head position

Personal View	Shared View
users can interact with the model at the same time	users can interact with the model at the same time

Instead of separating user interaction as input to the Shared and input to the Personal View based on number of fingers touching the screen
Having the option to edit a model from both view zones will give the necessity to determine which touch event corresponds to the Personal View zone and which corresponds to the Shared View zone. Calibration of the camera is necessary to achieve this function. Boundaries between the Shared View zone and the Persona View zone have to be set in the 3D coordinate system of the camera. The view zone the user is situated in can then be found. When the user situated in this zone inputs a touch command, this command will be forwarded to the SketchUp application. When the user moves into the Personal View zone the camera will detect this position. It will then signal the application listening for touch events to send the commands to the Ogre renderer running the Personal View application. Furthermore, in the situation with different users the camera will be able to detect which of them are in the Shared View zone and which are in the Personal View zone and display the Personal Overlays of the latter.

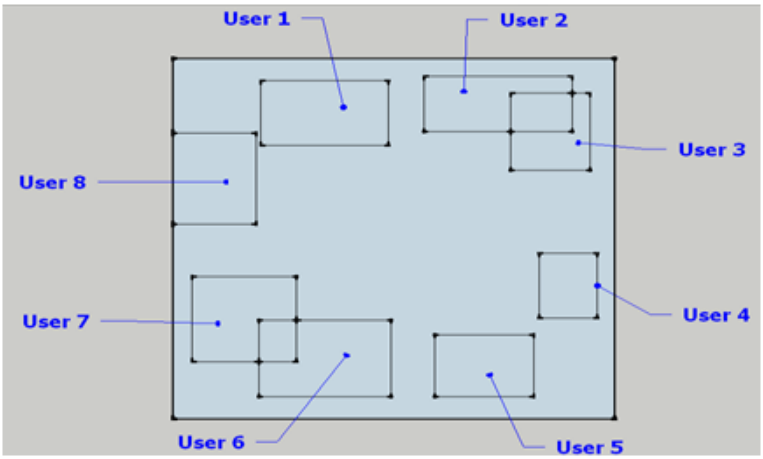
6.1.4 User Collaboration on the Screen

As discussed above instead of having a single user of the Shared View change his position to see the

Personal View more than one person users can be situated around the tabletop display. One of them could be working on SketchUp and the other ones viewing the rendered model and discussing it. A single user at a time will be able to modify the model either from the Personal or from the Shared View zones. If the touches made on the touchscreen by one user could be distinguished from the touches made by another user, then more than one user could work on the same model. This can be accomplished by using a depth camera such as the one in Microsoft Kinect. Kinect determines the head position of the person the finger touching the tabletop corresponds to using a depth cue. It will be able to separate the touches received by different people by matching a touch to the user wearing the head tracker situated at the users head coordinates supplied by Kinect.

Personal View	Shared View
users can interact with the model at the same time	users can interact with the model at the same time

Adding this feature to the interface will enable more than more than one person to work on the model without occluding the Personal View with his work. Task separation will be implemented using these overlays as each user will only see one particular overlay.
This can be very useful to give different users the right to access and modify certain elements of the whole object.



6.2 Real-time View Update

More research and development can be done in increasing the update speed between the two applications – Shared and Personal. Mesh exporting can be dealt with by parallel processes. Different threads can be made responsible for exporting the different parts of the model – texture, material and mesh. This can be done using the Ruby API which supports multithreading.

The export process will work faster if only the modified submeshes of the SketchUp model are exported. This will allow for export to take place at smaller time intervals because exporting a piece of the model will take less time than exporting the whole model. The exporter needs to be made

flexible so that it allows the necessary amount of time for the export of a mesh or submesh depending on the amount of submeshes the user modified. Furthermore, this solution can be extended into having different submeshes being exported by different threads concurrently. With a good processor and multiple core processors the export speed will increase making the SketchUp model to rendered mesh update speed closer to real-time. The import procedure in Ogre can remain the same or change to improve the performance even more.

6.3 Limitations

The update speed for the second time frame (updating the Personal View) is not real-time. This is a result of the export speed of large and complex models using a Ruby script simultaneously to the application thread run by SketchUp. If the export script was running concurrently with the thread, the export speed would increase. At the moment this is not possible. Even running simultaneously to the main process, however, the exporter will not be able to quickly convert large SketchUp models into meshes that can be loaded by Ogre. The solution for this problem is using machines with bigger CPU capabilities. With this in equipment the export can be sped up by allowing more computational resource for the OgreXMLConverter. Currently the CPU and memory usage of the Converter ranges from [RANGE]. The script itself processes the models in a few steps before running the Converter. If these steps are run concurrently in different threads the processing speed will increase. This will make it possible to make use of machines with several CPU cores to process the models and XML files. Currently, the Ruby implementation supported by the SketchUp plugin development system does not support multithreading.

Conclusion

This project is based on years spend by researchers to develop the necessary display design and implement its features.

To conclude this thesis, it has to be mentioned that the developed interface is one small but necessary step towards the development of more advanced interactive systems that support the creation of 3D graphics for the purposes of animation production and architecture.

Touchscreen displays supporting 3D design have already become popular and it is a matter of time before the whole animation studios start working on them only and integrate multi-view systems in their work.

References

1. TouchLight: an imaging touch screen and display for gesture-based interaction. Wilson, A. D. s.l. : In ICMI (2004), 2004.

2. A practical multi-viewer tabletop autostereoscopic display. Ye, G., State, A., Fuchs. 2010.

3. Multi-user, multi-display interaction with a single-user, single-display geospatial application. Karnik, A.,. 2011.

4. Interactive stereoscopic display for three or more users. Kitamura, Y., Konishi, T., Yamamoto, S. and Kishino, F. s.l. : In SIGGRAPH 2001, 2001.

5. Glasses-free 3D viewing systems for medical imaging. Daniel S.F. Magalhães, Rolando L. Serra, André L. Vannucci, Alfredo B. Moreno, Li M. Li. 2011.

6. 3D Visual Component Based Approach for Immersive Collaborative Virtual Environments. Okada, Y. 2003.

7. E-CONIC: A PERSPECTIVE-AWARE INTERFACE FOR MULTI-DISPLAY ENVIRONMENTS. NACENTA, M. A., SAKURAI, S., YAMAGUCHI, T., MIKI, Y., ITOH, Y., KITAMURA, Y., SUBRAMANIAN, S. AND GUTWIN, C. s.l. : IN PROCEEDINGS OF THE 20TH ANNUAL ACM SYMPOSIUM ON USER INTERFACE SOFTWARE AND TECHNOLOGY, ACM,(2007), 2007.

8. Multi-user, multi-display interaction with a single-user, single-display geospatial application. . Forlines, C., Esenther, A., Shen, C., Wigdor, D. and Ryall, K. s.l. : In UIST '06 2006, 2006.

9. Design for individuals, design for groups: tradeoffs between power and work-space awareness. . Gutwin, C. and Greenberg, S. s.l. : Proceedings of the 1998 ACM con-ference on Computer supported cooperative work, ACM,(1998), 1998.

10. Beyond "social protocols": multi-user coor-dination policies for co-located groupware. Morris, M. R., Ryall, K., Shen, C., Forlines, C. and Vernier, F. s.l. : In Proceedings of the 2004 ACM conference on Computer sup-ported cooperative work, ACM, 2004, 2004.

11. Introducing Maya 2011. Derakhshani, Dariush. 2011.

12. SandCanvas: A Multi-touch Art Medium Inspired by Sand Animation
Rubaiat Habib Kazi, Kien-Chuan Chua, Shengdong Zhao, Richard C. Davis, Kok-Lim Low

13. Open Sound Control - A Flexible Protocol for sensor networking" (Freed, Schmeder, Zbyszynski)

14. Multiview User Interfaces with an Automultiscopic Display
Wojciech Matusik, Clifton Forlines, Hanspeter Pfister, 2008

15. A MULTI-TOUCH 3D SET MODELER FOR DRAMA PRODUCTION
Maarten Cardinaels, Karel Frederix, Johan Nulens, Dieter Van Rijsselbergen, Maarten Verwaest, Philippe Bekaert, 2008

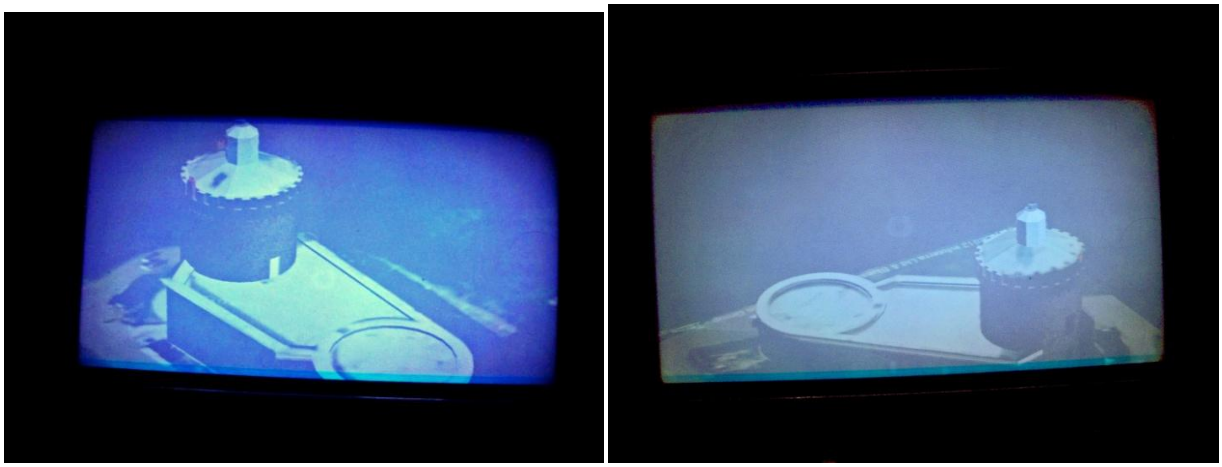
To this end, we developed an integrated modeling tool for drama production. This tool allows directors to pre-visualize drama scenes in 3D using an intuitive multi-touch interface. Multi-touch technology allows a very intuitive and user-friendly interface for manipulating 3D scenery

16. A Novel Multitouch Interface for 3D Object Manipulation

17 Distributed Rendering for Multiview Parallax Displays
Annen T, Matusik W, Pfister H, Seidel H-P, Zwicker M

Appendix

Personal View – touch interaction response



Shared View