A Zoomable 3D User Interface using Uniform Grids and Scene Graphs

Master Thesis in Computer Science

Vidar Rinne
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

November 25, 2011

Supervisor:
Thomas Larsson
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
Abstract

Zoomable user interfaces (ZUIs) have been studied for a long time and many applications are built upon them. Most applications, however, only use two dimensions to express the content. This report presents a solution using all three dimensions where the base features are built as a framework with uniform grids and scene graphs as primary data structures. The purpose of these data structures is to improve performance while maintaining flexibility when creating and handling three-dimensional objects. A 3D-ZUI is able to represent the view of the world and its objects in a more lifelike manner. It is possible to interact with the objects much in the same way as in real world. By developing a prototype framework as well as some example applications, the usefulness of 3D-ZUIs is illustrated. Since the framework relies on abstraction and object-oriented principles it is easy to maintain and extend it as needed. The currently implemented data structures are well motivated for a large scale 3D-ZUI in terms of accelerated collision detection and picking and they also provide a flexible base when developing applications. It is possible to further improve performance of the framework, for example by supporting different types of culling and levels of detail.
# Table of Contents

1 Introduction 4  
2 Related Work 4  
3 The Framework 6  
   3.1 Using the Framework 6  
      3.1.1 How to Start 6  
      3.1.2 Scene Graph Example 7  
      3.1.3 Rendering Example 7  
   3.2 Algorithms & Data Structures 8  
      3.2.1 Picking 8  
      3.2.2 Uniform Grid 8  
      3.2.3 Scene Graph 9  
      3.2.4 Collision Detection 11  
   3.3 Other Functions 14  
      3.3.1 File Formats 14  
      3.3.2 Shaders 14  
      3.3.3 Text 14  
      3.3.4 Texturize 14  
4 Examples 14  
   4.1 Photo Viewer 15  
      4.1.1 Features 15  
      4.1.2 Algorithms & Data Structures 18  
   4.2 Scene Viewer 19  
      4.2.1 Models 19  
      4.2.2 Modeling 19  
   4.3 Floating Controls 20  
   4.4 Texture Blending 21  
5 Results 21  
   5.1 Performance 21  
   5.2 Pilot Study 22  
6 Conclusions & Future Work 23  
7 References 27
1 Introduction

Zoomable User Interfaces (ZUIs) also called Multi-scale Interfaces are very common and many developers are building systems based on them [12]. A ZUI is designed so that users can see an overview of the content in the application which is achieved by the ability to zoom [18]. The zoom effect can be made in different ways, for example zooming a specific object or zooming the entire view [12]. Objects can also be replaced with different level of details at different zoom levels [12]. Many ZUIs are made using two dimensions such as ZoneZoom and PhotoMesa [5, 7, 20] but there are a few known three-dimensional zoomable user interfaces such as BumpTop [1] and Google Earth [9]. Zoomable interfaces can hold and view many objects at the same time which makes performance a problem, for example rendering speed when many objects are visible but also collision detection between objects. Another known problem with ZUIs is that users can get disoriented when navigating and not knowing how to use the interface to get back to the recent state, Stuart Pook et al. are describing these users as “lost in hyperspace” [18, p1]. Other evaluations [1, 5, 7, 17] report that users liked the interfaces in different ways.

The purpose of this project is to explore how 3D-ZUIs can be built, how flexible they can be and what they can be used for. The base features of a ZUI were implemented into a framework on which several example applications were built. The framework was developed to support the basic features and features needed for the example applications including the two major data structures used in this project which are the uniform grid and the scene graph, both are explained in the following sections. The example applications are a photo viewer demonstrating features like zooming, throwing and collision detection using a uniform grid, a scene viewer demonstrating the strengths of the scene graphs, a floating control example with event handling and node search and at last a multitexturing viewer demonstrating the strengths of the scene graph, both are explained in the following sections.

This report is structured as follows: chapter two presents related work on scene graphs, uniform grids and 2D- and 3D-ZUIs. Chapter three presents the framework and its capabilities including algorithms and data structures. Chapter four describes the example applications developed during this project. Chapter five presents the results including theoretical performance and pilot study. Chapter six presents conclusions and future work. The seventh and final chapter contains a list of all references used in the project.

2 Related Work

A common topic for uniform grids is ray tracing as the purpose of the grid is to divide space and related data into partitions to reduce intersection tests [10, c7, p285]. The grid is used to reduce the ray/object intersection tests for a ray in 2D- or 3D-space. By registering objects into partitions and traversing the grid it is possible to faster determine nearby objects. Common algorithms used for ray traversal in a grid are often based on a Digital Differential Analyzer (DDA) such as “A Fast Voxel Traversal Algorithm for Ray Tracing” presented by John Amanatides and Andrew Woo [3]. It is described as a simple high performing algorithm that needs only a few comparisons for ray traversal and it also requires little memory.

F. Cazals et al. presents a solution for ray-tracing using uniform grids in the paper “Filtering, Clustering and Hierarchy Construction: a New Solution for Ray-Tracing Complex Scenes” [11]. The solution of using a hierarchy of uniform grids and filtering and clustering of objects is aimed for more complex scenes. Several structures such as bounding volume hierarchies, BSP trees, uniform grids, subdivided uniform grids and octrees are discussed in the paper. During the analysis the author brings up the uniform grid as is having optimal performance with equally scattered objects but the structure will not handle non-uniform distribution very well. The author claims that experiments are required to find the best trade-off of "cost of intersections" vs. "cost of traversal". A comparisons between recursive grids and hierarchy of uniform grids shows that recursive grid may explode in memory consumption and that it is easier to handle this issue in hierarchal uniform grids. In octrees (special case of recursive grid) a voxel is always divided into eight new voxels and the author is describing the tree as unpredictable since it can get very deep and may contain many duplicates.

Uniform grids are also used to optimize performance in games [25, p249, p385]. Performance demanding algorithms such as collision detection and picking are accelerated using the grid when there

---

2The OpenGL Extension Wrangler Library: (glew.sourceforge.net) Accessed: 23112011
3The OpenGL Utility Toolkit: (www.opengl.org/resources/libraries/glut) Accessed: 23112011
4Simple DirectMedia Layer: (www.libsdl.org) Accessed: 23112011
are many objects in the scene. Picking uses ray traversal to find intersecting objects while collision detection is searching for nearby objects by examining the partitions the object currently is located in. It is also possible to load parts of the map or world at the time depending on an objects location in the grid and to play sound from objects located around a specific location.

A toolkit called IRIS Inventor presented in the paper “IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics” written by John Rohlf and James Helman [21] was developed with real-time 3D graphics and performance in mind. The primary data structure of the toolkit is a hierarchical scene graph which is built of several node types such as scene-, switch-, LOD-, light and group nodes. Some specific purposes of the scene graph are that its geometric data structures are optimized for using immediate mode which claims to be a good choice for animations, the scene graph is also constructed to reduce graphics mode changes for faster graphics rendering and also to allow parallel traversal for multiprocessing capabilities. Some other speed up techniques are culling, bounding volume hierarchy and level of detail.

Another toolkit presented in the paper “An Object-Oriented 3D Graphics Toolkit” [23] written by Paul S. Strauss and Rikk Carey is focusing on interactive 3D applications. It is presented as a general, object-oriented and extensible framework aimed for developers. Another goal of this toolkit was to make it interactive and to be able to interact directly with 3D-objects which was not very common at the time. The toolkit was built using an object-oriented approach using scene graphs for dynamic object representation. Some node types are transform-, camera-, light-, material-, and several types of shape nodes. Bounding volumes and event handling were implemented to make the toolkit support picking. Actions like RayPickAction can be used to get the most front object and more information can be collected by using queries. Picking is accelerated by separator nodes which for example can be used to cache bounding boxes for faster culling and real-time highlighting of objects. Another feature that was implemented was the track ball for rotation of objects. It was represented as an interactive sphere encapsulating 3D-objects. Other performance improvements are tuned rendering paths, cached states by storing information in separator nodes and display lists.

The paper “The Data Surface Interaction Paradigm” written by Rikard Lindell and Thomas Larsson [17] presents a solution to make interaction easier by developing a prototype live music tool based on 2D-ZUIs. The application was implemented with features like incremental search, zoom, pan and other functions like visual feedback and text completion. The application was evaluated by ten test users and responses were that it felt free and was fun and easy to use [17]. Hierarchical scene graphs were used to represent the components which include information like position and scale. The graphics library used was Simple Direct Media Layer (SDL) based on the OpenGL API along with the FreeType font library that offers texture mapped fonts.

ZoneZoom is a technique for handling input on smart phones. It was developed because of navigation difficulties on current smart phones. In the paper written by Daniel C. Robbins et al. [20] they give an example of using their technique on a map navigation software where the keys on the keypad are mapped to nine different grid cells on the map, for example key 4 takes the user to the map cell to the left while 6 to the right, map cell 5 is always in the middle of the screen and allows the user to zoom in on the map part that is currently under the cell. Their conclusion brought up that this system has been adapted to a map monitoring application which has been deployed to over 1000s of users.

Desktop PhotoMesa is a zoomable photo browser which is described in the paper by Benjamin B. Bederson and Amir Khella [5, 7]. PhotoMesa uses tree maps to place pictures on the screen. Pictures in directories are placed in groups. When clicking a specific group the application zooms in on the directory. Low resolution pictures are used as thumbnails but full resolution while zoomed. There are also possibilities to navigate on the screen using the keyboard. The application was implemented in JAVA with the Jazz toolkit [6] which can help building zoomable two-dimensional interfaces using scene graphs. Feedback from users using the pocket version of the PhotoMesa did not notice any increase or reduction in time when compared to software that did not have the same features. Users reported that PhotoMesa was more fun than alternatives and also easy to use.

In George W. Furnas and Xiaolong Zhang’s paper [12] they present an application called MuSE (MultiScale Editor). It is written in Td/Tk and Pad++5 which is software for creating zoomable interfaces. The operating system was UNIX. The purpose of this application was to examine several zooming techniques which are presented in the paper, for example bounded-geometric-zooming which means that the object cannot be larger or smaller than specified boundaries. If too large they will become invisible not to cover other objects and If

---

too small they can be filtered away. Another simple zoom effect that basically works in the same way is fade-bounded-geometric-zooming which fades the object when it is getting too big or too small on the screen. Fixed-size-zooming gives the effect that the object is fixed but the environment is zoomed. Semantic zooming is also discussed which means that objects are replaced while zooming. More zooming effects can be found in George W. Furnas and Xiaolong Zhang’s paper “MuSE: A MultiScale Editor” [12]. Human Computer Interaction Lab is no longer developing or supporting PAD++ but is still exploring ZUIs and is currently working on a new JAVA-based toolkit called Piccolo6.

The 3D-desktop software called BumTop presented in the paper “Keepin’ It Real: Pushing the Desktop Metaphor with Physics, Piles and the Pen” written by Anand Agarawala and Ravin Balakrishnan [1] aims at making the desktop more physically correct. It is considered an alternative for regular desktops. The interaction is optimized for using a pen where 3D-icons easily can be moved, thrown as well as selected by lasso tools. Some important design goals presented are smooth transitions and piling. Smooth transitions means that animations are smooth, for example when dragging an object it will increase and decrease in speed instead of clamping to the cursor. Piling means that icons can be on top of each other instead of side by side. It is also possible to browse between the piled objects and to get an overview. The software is using a physics library that takes care of collisions, although physics can be deactivated if necessary. The evaluation shows that users liked the realistic feel, features like throwing/tossing and the smooth transitions. Users also reported that they felt familiar with the software and most people completed the training without extensive practicing.

Google Earth is another navigation tool that provides many services [9]. It is using a 3D-globe with satellite images as surface where roads and other choices can be highlighted through settings. Most common places and cities are modeled in 3D. Most necessary functions were implemented into the framework to support the features in the example applications. The uniform grid and scene graph related features got the highest priority. It is also possible to work from any angle. Microsoft Surface aims at many areas such as IT, education, financial and health care among others. It is a way to connect people, to learn, entertainment and work. Microsoft Surface 2.0 runs on top of Microsoft Windows 7 and has 360 degrees of freedom. It is optimized for over fifty simultaneous touch points and has object recognition of objects placed on the screen. It is possible to develop custom applications by suppressing the Windows 7 interface and use your own touch user interface.

3 The Framework

A framework was developed during this project to support basic 2D/3D-operations and viewing capabilities. It is developed with support from a uniform grid which is a partitioning structure and the scene graph for flexible models. The main purpose of the uniform grid is to speed up features like picking and collision detection while the scene graph structure is mainly used for flexible 3D-models. The framework is written in C++ using Microsoft Visual Studio and GLEW to get necessary information and at last GLUT and SDL as graphics libraries. The most necessary functions were implemented into the framework to support the features in the example applications. The uniform grid and scene graph and related features got the highest priority.

This section is structured as follows: the first subsection shows some examples of how to use the framework. The second subsection describes the most important algorithms and data structures and the third subsection presents other functionality.

3.1 Using the Framework

There are a few steps necessary to get the features of the framework available. Some example code about how to set up the framework, the scene graph and how to render a scene are described below.

3.1.1 How to Start

To be able to use functions provided by the framework the base class must be inherited into a custom class. Below is a piece of code describing the custom class my3DZUI that inherits the properties of the ZUI_App class:

```java
class my3DZUI:public ZUI_App{
...
};
```

---

Before the underlying data structures can be used the ZUI App must be initiated. It is achieved by calling the init function which takes the x- and y-resolution and a Boolean value whether the application should be set to full screen or not. If the window caption is to be set it must be called after the init function.

```c
ZUI_Init(1920, 1080, true);
ZUI_SetCaption("Zoomable User Interface");
```

How to allocate and start an instance of the new ZUI class from the main thread is shown below. The pointer `z` becomes the handle to an instance of the custom 3D-ZUI class and the main loop is started by calling the loop function.

```c
int main(void) {
    my3DZUI* z = new my3DZUI();
    z->Loop();
    delete z;
    return 0;
}
```

### 3.1.2 Scene Graph Example

Below is a demonstration on how to set up a scene graph and how to add it to the scene. The first step is to create a SceneGraph object:

```c
SceneGraph* sg = new SceneGraph();
```

To be able to render a geometric object there is a node called GeometryNode which holds a mesh. The mesh object contains vertices, normals and texture coordinates which are used when rendering the model. A mesh can be loaded using the 3D-Studio Max file format (.3ds).

```c
GeometryNode* gn = new GeometryNode();
Object* o = ZUI_LoadMesh("c:/mymesh.3ds");
```

Surface properties can be set by using the MaterialNode. Below is a JPEG image loaded from some location and then converted into a texture. Shaders can be used to further change properties of the material, for example effects such as bump mapping, lightning methods such as phong lightning, cartoon and much more. Shader parameters can be set to change effects dynamically.

```c
GLuint tex;
int texWidth, texHeight;
MaterialNode* mn = new MaterialNode();
ZUI_Texturize("c:/mypic.jpg", tex,
    &texWidth, &texHeight);
mn->InitializeTexture(tex, 0, "texture1");
mn->InitializeShader(sc->GetShaderObj(0)
    ->shaderProgramID);
```

Position, rotation and scaling of geometric objects can be set by adding nodes of the type TransformationNode. The first transformation node in the tree is placing the object in world space. Transformation nodes further down the tree inherits the parent node’s coordinate system.

```c
TransformationNode* tn=new TransformationNode();
tn->setScale(1.0, 1.0, 1.0);
```

The next step is to connect the individual nodes into a tree with the SceneGraph as top node. The transformations must be applied before geometries. Material can be applied before and after a transformation node but only before the geometry node.

```c
gn->AddMesh(o);
mn->AddChild(tn);
tn->AddChild(gn);
sg->SetRoot(mn);
```

Finally the newly created scene graph is added to the scene. For interaction it must also be inserted into the grid (optional).

```c
ZUI_AddToScene(sg);
ZUI_GetGrid().Insert(sg);
```

### 3.1.3 Rendering Example

The code below shows how a possible rendering process may look like. It is important that all rendering code is placed between the begin-scene and end-scene calls.

```c
ZUI_BeginScene();
If the scene contains movable objects with or without collision detection they must be updated.
ZUI_Update();
```

After the objects position and other parameters have been updated they can be rendered.

```c
ZUI_RenderScene();
ZUI_RenderGrid();
```

Menus, text and other 2D-graphics can be drawn using 2D-mode. It is important to disable the 2D-view when finished, otherwise it will affect the entire rendering process.

```c
ZUI_Enable2DView();
fileMenu->Render();
clickMenu->Render();
ZUI_RenderString(text, &infoRect);
ZUI_RenderCameraBorders();
ZUI_RenderCustomMouseCursor();
ZUI_Disable2DView();
```
And finally the end-scene call that tells the framework that the rendering chain is complete.

\[ \text{ZUI\_EndScene();} \]

### 3.2 Algorithms & Data Structures

The framework provides functionality by using several major algorithms and data structures. The most important ones are described below such as picking and collision detection algorithms and the uniform grid and scene graph structures.

#### 3.2.1 Picking

Today there are many known picking techniques. One of them is achieved by using OpenGL’s selection mode that basically uses a name stack that needs to be pushed and popped and the hits get stored in a list \[4, c3, p124-125] [8] [10, c4, p78]. Another picking technique uses a buffer in OpenGL. By rendering objects in different colors while lighting and effects are turned off to a second buffer it can be examined if an object is visible at different points by examining the color \[4, c3, p124-125] [8] [10, c4, p80]. This technique can also be improved in terms of speed by using bounding volumes instead of the real objects, although precision may vary [10, c4, p81]. A third picking technique which is used in this project is ray casting. It means that a ray is sent out in 3D-space from the click or shooting point, which for example can be done from the mouse cursor. This technique is similar to a bullet path in a game or a ray in a ray tracer [24]. The motivation for the ray casting technique is that an object doesn’t have to be rendered twice or go through the OpenGL pipeline. By combining a bounding volume hierarchy (BVH), the scene graph and the uniform grid it is possible to greatly reduce intersection tests.

#### 3.2.1.1 Algorithm

Below are the steps necessary to perform picking on an object. The whole procedure goes through the uniform grid and the scene graph structure to be complete.

1. Create a ray from mouse position.
2. Transform ray to world coordinates. (same as the uniform grid coordinate system)
3. Decide in which grid voxel the starting point is. (picking/grid algorithm)
4. If voxel has any scene graph connected to it, traverse scene graphs:
   - Transform scene graph BV to world coordinates. (grid coordinates)
   - Ray intersection test with BV. (picking/scene graph algorithm)
5. Traverse grid in ray direction. (this is part of grid algorithm)

#### 3.2.2 Uniform Grid

A grid is constructed by cells or in some cases called voxels. The grid is uniform if all voxel sides are of the same length (Figure 3.1 – 3.2). An application with many objects requires many computations, too many for computers to handle in real time. A grid is used to divide the scene into groups and by inserting objects into voxels the computations can be limited to objects that are more likely to collide [10, c7, p285]. For example in picking where the interesting objects are registered in voxels that are intersecting the ray. Collision detection only needs to check for collision by searching for objects in nearby voxels. A grid with many small voxels requires a great number of computations during ray traversal and collision detection. This framework uses a uniform 3D-grid with a traversal algorithm that is a variant of the 3D Digital Differential Analyzer (3D-DDA) to reduce the number of voxels visited during picking.

![Figure 3.1 - 3.2 – A 3-dimensional uniform grid.](image-url)
3.2.2.1 3D-DDA Traversal Algorithm

The algorithm for traversing the uniform grid is a DDA-algorithm by John Amanatides and Andrew Woo [3]. The algorithm basically breaks down a ray into segments where each of these segments spans one voxel. The process includes two phases. The first part handles the initialization and the second part handles the incremental traversal. Note that the framework is using a three-dimensional version of the method described. It is however simple to extend the algorithm to handle three dimensions. Figure 3.3 shows an example of which voxels that needs to be traversed along a ray.

```
loop {
    if(tMaxX < tMaxY) {
        tMaxX = tMaxX + tDeltaX;
        X = X + stepX;
    }
    else {
        tMaxY = tMaxY + tDeltaY;
        Y = Y + stepY;
    }
    NextVoxel(X, Y);
}
```

3.2.2.2 Grid Linking

When inserting a scene graph into the uniform grid it is passed to an insert function located in the uniform grid class. Insertion into grid is required for picking and collision detection to work. The grid holds the information on how to insert the scene graph. When a scene graph is passed to the grid it will be traversed until the first bounding volume is found. The bounding volume will then along with all transformations be used to register the scene graph object into the grid voxels that are intersected by the bounding volume (see Figure 3.4).

```
loop {
    if(tMaxX < tMaxY) {
        tMaxX = tMaxX + tDeltaX;
        X = X + stepX;
    }
    else {
        tMaxY = tMaxY + tDeltaY;
        Y = Y + stepY;
    }
    NextVoxel(X, Y);
}
```

3.2.3 Scene Graph

A scene graph in general is basically a tree structure that is used to hold information of a virtual 3D world including its subparts [13]. It often includes components such as geometry-, transformation- and material nodes but also bounding volumes [21, 23]. The scene graphs predecessor the display lists was basically able to group coordinates to a single model, evolved versions were able to group several
models and later became scene graphs [22]. The idea of scene graphs was to add flexibility as well as a quicker way for implementation [19] and also to increase performance in some situations [22]. The motivation for using scene graphs in this project is that the scene graph data structure describes the information in an abstract and object oriented way [19] which makes it more flexible in terms of design, maintenance and components extensions. By using bounding volumes various speedups can be implemented for collision detection and picking. It is also possible to implement rendering speedup techniques directly in the scene graph like Level Of Detail (LOD) and culling [13, 19, 21, 23]. Although there can be performance issues if nodes have to be updated often due to traversal algorithms if the scene graph is complex [22].

Today there exists several scene graph libraries such as OpenInventor, OpenSceneGraph and OpenSG [14]. OpenInventor is simple to use and requires little code to set up. The book Inventor Mentor with tutorials for beginners is available. With OpenSceneGraph the programmer assembles each node from lower level components. The programmer also manages thread safety which means more work and code. QuickStart Guides with code examples for experienced programmers exists. OpenSG requires the programmer to assemble each node from lower level components and also to handle thread safety issues. It is single threaded and requires more code. QuickStart Guides with code examples are available for experienced developers. These three scene graph libraries have been tested with a high polygon model by David Harrison in the paper “Evaluation of Open Source Scene Graph Implementations” [14]. The results show that OpenSG performs best, OpenSceneGraph on second place and OpenInventor on third place. The author explains that the differences in performance possibly are due to the level of culling performed in the scene graphs.

Initially the decision was made to design a new scene graph for this framework. The reasons were uncertainty about compatibility between existing scene graph libraries and the uniform grid but also about performance and implementation. The scene graph in this project is constructed by several types of nodes. The root or top node which can be called a graphical component is implemented as the SceneGraph class. It contains critical information about the scene graph including functions to delegate information to and from child nodes. The defined nodes that are based on the ones in the book by Edward Angels [4, c10, p521-536] are described in the following subsections.

3.2.3.1 SceneNode
This class is not used as a node itself. As in Figure 3.5 all other nodes inherit the properties from this node so they can be handled equally when traversing the tree. This node defines connections, basic parameters and functions that are the same for all nodes such as pure virtual functions.

3.2.3.2 GeometryNode
This is a node class that holds the geometry in the scene graph. It includes vertices, normals, texture coordinates and bounding volumes for the current model. A render function to draw the object to the screen is also defined in this class. This node should be the leaf of the tree and all materials and transformations should have been applied before rendering the geometry.

3.2.3.3 TransformationNode
This class holds transformation parameters such as position, rotation and velocity which can be set at any time. The rendering traversal is recursive and will pass the node twice. Every time a transformation node is passed it loads the current transformation matrix into OpenGL and on the way back it will restore the transformations as they were before. This means that children nodes inherit the current transformation from the parent node. A transformation node is a group node and can hold unlimited number of children of any type. It is also holding a BV that encapsulates the childrens BVs.

3.2.3.4 MaterialNode
A material node contains information of how the surface looks on a mesh model. The node holds an id for which shader and texture to use but also a shininess parameter that is responsible for the size of the highlights caused by nearby lights. Other parameters that are used when no texture is assigned are specular, diffuse and ambient color. Specular has a relation to highlights, diffuse is the base color and ambient is the base light.
3.2.3.5 Traversal Algorithms

The traversal algorithm that is used to render the scene graph is a pre-order or depth first algorithm (see Figure 3.6). A pre-order algorithm means that the left side of every node is traversed first and the right side last, which is the same as depth first. This algorithm is recursive and is built to apply properties when traversing down the tree and restore properties on the way back. Transformation and material nodes can be connected in any technically but a geometry node can only be set as a leaf node which always is on the last position in the current sub tree. Let’s assume that the TransformationNode T connected to the scene graph is the first node in the tree. All child nodes that are rendered inherits the transformations of T, the G nodes will be rendered with different materials.

Imagine that the GeometryNode G in the left subtree in Figure 3.6 is a very large donut covered in shiny chocolate material M. The G in the right subtree is a plate with a light non-shiny looking material M. The first transformation node T in the scene graph represents the position of the entire object. If T is moved or rotated the nodes in the subtree will follow, including the geometry nodes. Below is an example of a rendering traversal algorithm of a scene graph.

1. Enter the root node T in the scene graph and apply current transformations.
2. Traverse the left child node of T which is M and apply material properties (step 1).
3. Traverse the child node of M which is G and render the geometry (step 2).
4. Traverse the parent node of G which is M and restore various material parameters (step 3).
5. Traverse the parent node of M which is T and continue to the next connected child (step 4).
6. Traverse the right child node of T which is M and apply material properties (step 5).
7. Traverse the child node of M which is G and render the geometry (step 6).
8. Traverse the parent node of G which is M and restore various material parameters (step 7).
9. Traverse the parent node T and continue to the next connected child if there is any, otherwise return and exit the function (step 8).

Figure 3.6 - Rendering traversal example of a scene graph.

Figure 3.7 below shows an example of how the tree is traversed during picking. The group can be seen as a list with objects. PBV is a Parent Bounding Volume and CBV is a Children Bounding Volume. Two objects are compared against a picking ray. The left side of the tree shows a successful hit on the parent volume and it continue to examine if its children are hit. The left side shows an unsuccessful hit on the parent BV and the children BVs are not tested at all.

Figure 3.7 – Bounding volume traversal.

Note: Group = some object list, PBV = Parent Bounding Volume, CBV = Child Bounding Volume

3.2.4 Collision Detection

A feature of the provided by the framework is that it is possible to move objects by hitting them with other objects. To reduce the number of computations only the necessary objects are tested. It can be done by using a space partitioning structure such as a grid (see Figure 3.8). By connecting objects (scene graphs) to the intersecting voxels of a uniform grid it is possible to limit collision detection by testing collisions on objects registered to nearby voxels instead of all other objects in the scene.
An early version of the framework used Axis Aligned Bounding Boxes (AABBs) for object/object intersection tests due to very fast collision detection and for rapid prototyping. Later on the scene graph structure was extended to support rotations and the decision was made to use Oriented Bounding Boxes (OBBs). OBB/OBB intersection tests require more computations than AABB/AABB but can handle collisions with non-aligned boxes correctly which is impossible with AABB/AABB methods.

The following subsections describe the bounding volumes in this project including related algorithms and data structures.

### 3.2.4.1 Axis Aligned Bounding Boxes

A common data structure in computer graphics is the axis aligned bounding box. The bounding box is used to encapsulate an object which often has a more complex shape. It is used mainly because of simple and fast intersection tests for both picking and object/object collision detection. Axis aligned means that the axes of the box are parallel to the world coordinate axes, or in other words have the same coordinate system.

Three AABB structures with intersection tests are described in the book Real Time Collision Detection [10, c4, p78] where the AABB structure using a min point \((a^{min})\) and a max \((a^{max})\) point are used (as illustrated in Figure 3.9). The extreme points are used to insert objects into a grid by transforming the coordinates to world coordinates and then decide in which grid voxels the min and max point are. All voxels covered by the AABB will be connected to the object by using a simple loop. The structure is not the most memory conserving representation of an AABB because of the six floating point values [10, c4, p79] but it is easy to work with and is excellent for fast development and testing purposes.

\[
\text{AABB} \{ \text{Point3d } \text{min}; \text{Point3d } \text{max}; \}
\]

### 3.2.4.2 Intersection Algorithms

There are two kinds of intersection tests that are used for AABBs in this framework. The first is the ray/box intersection test that is used for the picking feature. The second is the AABB/AABB intersection test used for collision detection between objects and when inserting objects into the grid. Both intersection algorithms are described in the following sections.

**AABB/AABB intersection**

This algorithm is described in the book Real Time Collision Detection [10, c4, p79]. It is based on the Separating Axis Test (SAT) [2, c16, p744-745] to decide whether there is an overlap or not. Below is the intersection tests needed for detecting collision between two AABBs expressed in code.

```c
int TestAABBAABB(AABB a, AABB b) {
    // Exit with no intersection if separated
    // along an axis
    if (a.max[0]<b.min[0] || a.min[0]>b.max[0])
        return 0;
        return 0;
        return 0;
    // Overlapping on all axes means AABBs are
    // intersecting
    return 1;
}
```

Figure 3.10 illustrates the separating axis method with two BVs in 2D-space. The BV’s sides are split into components \((x, y)\) by projecting them against the world axes. In this case there are two AABBs so the projection doesn’t need any specific calculations since the AABBs are already axis aligned. Only two axes needs to be tested in two dimensions to determine if there is an overlap or not. A collision succeeds only if the BV’s projected sides overlap on all axes. For three-dimensional AABBs it requires three intersection tests, one for every axis.
RAY/AABB intersection

The RAY/AABB intersection test is a method that is based on the separating axis test [10, c5, p179]. The original method is described below [2, c16, p744-745]:

The parameters \( c \) and \( w \) are two points representing a line. Variable \( h \) is the half vector of the box \( B \).

\[
\text{RayAABBOverlap}(c, w, B) \quad \text{returns} \quad \{(\text{OVERLAP}, \text{DISJOINT})\};
\]

\[
vx = |wx| \quad vy = |wy| \quad vz = |wz|
\]

if(|cx| > vx + hx) return DISJOINT;
if(|cy| > vy + hy) return DISJOINT;
if(|cz| > vz + hz) return DISJOINT;
if (|cy*wz - cz*wy| > hy*vz + hz*vy)
return DISJOINT;
if (|cx*wz - cz*wx| > hx*vz + hz*vx)
return DISJOINT;
if (|cx*wy - cy*wx| > hx*vy + hy*vx)
return DISJOINT;
return OVERLAP;

3.2.4.3 Oriented Bounding Boxes

An oriented bounding box can be seen as an extension of the axis aligned bounding box. A difference is that an OBB can be aligned in any direction and thus be able to encapsulate objects with less unnecessary space than an ordinary AABB as well as it can handle rotation. Figure 3.11 represents an OBB with three normalized \( u \)-vectors containing the current alignment of the box, a center point \( c \) and finally a vector that holds the half-lengths \((e_x, e_y, e_z)\) representing the size of the box along the \( u \)-axes.

3.2.4.4 Intersection Algorithms

There are two kinds of intersection tests necessary for object/object collision detection and picking to work with rotated objects. The first is the OBB/OBB intersection test which is very similar to AABB/AABB but has the capability of handling rotated boxes. The second intersection test is Ray/OBB which is used when selecting an object. The structure of the OBB can be seen in Figure 3.11.

OBB/OBB intersection

The collision detection between two OBBs is determined by SAT which is built on the same principle as for AABBs. OBBs can be aligned in any direction and therefore require more intersection tests (see Figure 3.12). Up to a maximum of fifteen axis tests are needed to be able to decide if two OBBs are colliding or not. To avoid the zero-vector issue in calculations a small constant called \( \text{epsilon} \) was used.

3.2.4.5 Ray/OBB intersection

The collision detection between a Ray and an OBB is done by checking if the ray intersects with the OBB’s surface. This can be done by checking if the ray intersects with the OBB’s six faces. If the ray intersects with any of the OBB’s faces, the collision is detected.

13
RAY/OBB intersection

As the framework supports rotated objects encapsulated by OBBs, the Ray/AABB intersection method becomes insufficient. Possible solutions are to either entirely rewrite the intersection method to detect all possible ways the ray can intersect the box or a more efficient solution where the OBB is translated into an AABB and then perform the same tests as Ray/AABB with the same number of operations. The second method which is used in this project requires that the position and direction of the ray are transformed with the box. The ray intersection points on the box before and after the transformations must be the same. This method is accomplished by placing the OBB in origo and rotates it by using the inverse of the rotation matrix. It is now aligned with the world axes. As the OBB was placed in origo the ray position must be subtracted by the distance between origo and the OBB’s center point. To get the correct direction of the ray it is rotated by the inverse of the OBB’s rotation matrix. The transformations are illustrated in Figure 3.13. The ray and OBB are now represented in world coordinates as a new ray and an AABB. The last step of the Ray/OBB intersection method is the Ray/AABB intersection method described in section 3.2.4.2.

Figure 3.13 – Ray/OBB represented as Ray/AABB.

3.3 Other Functions

3.3.1 File Formats

The supported format for loading 3D-models through the framework is the .3ds format. It is possible to export models directly from Autodesk 3D Studio Max and it is preferred to save the model while in mesh mode since it will force every polygon to be represented as triangles. This framework as well as many game engines only support triangle meshes. Exporting a whole scene is not supported in a single call. Only one connected mesh at the time can be loaded by the framework. The .3ds file format comes with vertices and texture coordinates, normals are not included but can be calculated through the framework.

3.3.2 Shaders

All objects can be styled with shaders, for example lightning methods such as per pixel phong lighting and surface effects such as bump mapping. A shader can contain several effects which makes it very flexible. It is also possible to pass parameters from an application to the shader for real-time effects. The motivation for using shaders is that they are highly programmable, easy swappable and that they can achieve a more lifelike and living environment in 3D-space. Shader effects can be used to point out the differences in depth and the form of the objects. The current limitation of how many textures that can be used simultaneously is the texture unit limit on the graphics card or OpenGL API. The framework supports one shader per geometry which is also easily swappable by using shader IDs (sID).

3.3.3 Text

The framework supports rendered text. By using SDL TTF (TrueType Font rendering library) text can be rendered onto a texture. It is possible to render text directly to a quadratic polygon on screen but also to render to a texture. Rendering to a separate texture can improve performance since the texture doesn’t necessarily have to be rewritten until the text changes. It is also possible to render text in 2D. By entering 2D-mode the framework sets up correct view matrices for 2D-rendering. If not using 2D-mode a texture rendered with text can be used on any 3D-object. All example applications in this project are using the Courier font (courier.ttf) of size thirteen with outlining.

3.3.4 Texturize

By using a function provided by the framework a picture can be loaded and converted to a texture in a single call. It is also possible to make a color transparent if needed. To be able to render textures with transparency the parameter ”GL_BLEND” must be enabled through the OpenGL API.

4 Examples

This section describes the example applications, their implemented features and algorithms and data structures. These examples are mainly developed to show the flexibility of the framework and what it is able to do in its current state. The first example is a small Photo Viewer. Its main purpose is to show a collection of the framework’s capabilities in a single application. The second example is a scene viewer. It contains larger 3D-models and uses
more depth then the previous example. The framework also supports events when clicking or hovering objects. This is shown in the floating controls example where every 3D-buttons changes something in the scene. They also have the ability to collide with other objects if moved. The fourth example is more graphical and technical example and includes a texture blended 3D-model.

4.1 Photo Viewer

The first example application developed in this project was the Photo Viewer. It is built upon the framework and its main purpose is to test a large collection of features as a 3D-ZUI. The application is designed as a Photo Viewer not very different from two-dimensional software. The biggest differences are in navigation, interaction with real 3D-objects, graphics with depth and real time effects, floating photos with the ability to collide.

4.1.1 Features

The currently implemented features such as arc ball, select and drag, grabbing, throwing, navigation, zooming and collision detection are explained in the following sections.

4.1.1.1 Arc Ball

A common implementation of an arc ball was integrated to be able to rotate objects, to decide the rotation of a sphere by calculating how two click rays intersects with a sphere. Figure 4.1 shows that A and B are rays (calculated from the mouse click), \( v \) and \( u \) are vectors from the spheres center to ray A and B. By calculating the cross product of \( u \times v \) we get the \( w \) vector which is representing the rotation axis. By using simple mathematics the angle \( a \) can be achieved which represents the rotation around the \( w \)-axis. An arc ball support rotations in any direction, any degree around every axis and it is possible to click on it from every position in 3D-space.

4.1.1.2 Select and Drag

An important feature is select and drag, it means that you can select a photo and interact with it such as moving the object by using the mouse cursor (see Figure 4.2).

4.1.1.3 Grabbing

Grabbing is a feature to ease the navigation when browsing photos. Simply grab the background (or grid) and drag it in any direction as in Figure 4.3 and then release as in Figure 4.4.
4.1.1.4 Throwing
A feature that was implemented into the Photo Viewer example application is throwing. Throwing is achieved by selecting and dragging a photo with the mouse and then releasing it while moving (see Figure 4.5–4.7). The photo will continue to move with initially the same speed as the mouse cursor but will later slow down due to a friction assigned to the transformation node of the objects. This feature was implemented to make interaction more real as a paper photo on a table. It can be useful to quickly move photos out of the way or to group them by throwing them to a specific place on the screen or usable area. If collision detection is enabled other photos will react differently depending on the speed of the colliding photos.

Figure 4.5 – Drag photo.

Figure 4.6 – Release photo while dragging.

Figure 4.7 – Photo now has its own velocity.

4.1.1.5 Navigation
One of the most critical features of a 3D-ZUI is the navigation system. There are various navigation techniques implemented. One of them is a steering technique called WSAD which navigates the camera in different directions, W is forward, S is backwards, A is left and D is right. Keyboard arrows and NUMPAD is also supported. It works in the same plane as the wall and supports eight directions incl. diagonal directions. NUMPAD 8 navigates the camera up, 2 down, 4 left, 6 right, 7 up left, 9 up right, 3 down right, 1 down left. To restore the view to the center there are two keys available, either C or NUMPAD 5 or by right clicking and selecting reset camera in the menu.

4.1.1.6 Zooming
As this application is built as a zoomable user interface it also has zoom capabilities (see Figure 4.8–4.11). The feature zooms towards the direction the mouse cursor currently is pointing at which makes it easy to select the area to enlarge. If you are zooming the upper left corner you will end up at the point you are pointing at. Zooming out at the same point will take you back along the same path. The zoom feature is also supported using the scroll button on the mouse and keyboard sign (+) zooms in and (–) zooms out. Double clicking on a photo it enters full screen 2D-mode with black background.

Figure 4.8 – Zoom level I.

Figure 4.9 – Zoom level II.
4.1.1.7 Screen Tracking

Screen tracking is another kind of navigation feature often seen in strategy and role playing games. By hovering tracking borders with the mouse cursor the screen will pan in the same direction (see Figure 4.12–4.13), for example hovering left tracking border it will take you to the left and by placing the mouse cursor in the upper left corner it will take you in that direction. This can be an optional navigation system to the keyboard or can be used at the same time. For example when a photo is to be moved outside screen space it’s possible to drag the photo over the tracking border and the view will pan in the same direction. The further the cursor is passing the borders the faster the screen will pan.

4.1.1.8 Colliding Photos

Another feature that was implemented was Colliding Photos which means that no photo can overlap another photo as long as collision detection is enabled. If you drag a photo into another the second photo will be pushed out of the way. If the second photo collides with another and another also they will be pushed out of the way, a chain reaction that is. If a photo no longer is grabbed or if it is thrown away it will react to collisions as any other photo. If a photo in some way is passing outside the wall it will be transferred to the opposite side automatically. This feature can be deactivated if necessary. Figure 4.14–4.16 are rendered screens from the Photo Viewer showing a high velocity object bringing chaos when causing a chain reaction.
4.1.1.9 Menus
The photo application uses menus. By right clicking with the mouse cursor in an empty space brings up a list of global setting that can be changed. Right clicking on a photo shows available functions for the specific object type, like rotation.

4.1.2 Algorithms & Data Structures
The following two sections explain the algorithms and data structures used in the Photo Viewer, features that is not fully supported or part of the framework.

4.1.2.1 Grabbing/Select and Drag
The Photo Viewer’s “grabbing” and “select and drag” feature are easy to implement for two-dimensional space, although for three-dimensions there are some differences. Due to the perspective view (non-orthogonal view) distances between points are not constant in three dimensions as in two. For example if you select and drag a photo that is near the screen the object will travel at the same speed as the cursor, which is similar to what happens in two dimensions. However, if you do the same with a photo far away it will need to travel much faster to keep up with the cursor. The problem is that the distances further away from the camera are not matching the distances between screen coordinates (mouse points) which is caused by the Field of View (FOV) value used to simulate the eye lens effect. The FOV parameter alters the view volume and makes objects further away smaller. Let’s take a look at the problem illustrated in Figure 4.17 and 4.18.

Figure 4.16 – Chain reaction resulting in chaos.

Figure 4.17 – Orthogonal view volume v1 where c1 and c2 are click rays.

Whether Figure 4.17 shows an example where the click rays c1 and c2 passing the orthogonal view volume v1 have the same distance when going in (d1) and out (d2), here would calculations for two dimensions suffice.

In Figure 4.18 it is shown that the distance between the rays grows the further away from the starting point it gets. Distance d3 is measured in the starting point and d4 is measured deeper into the volume, compared to no difference between d1 and d2 in Figure 4.17 the difference is huge. A larger field of view means greater distance between end points.

This issue can be avoided by using picking provided by the framework and setting up a virtual plane as described below:

Phase 1 - Click and hold:
1. Get the position the picture is located at (frameworks picking algorithm).
2. Use the vectors from camera and set up a parallel 2D-plane at the picture location.

Phase 2 - Hold and drag:
1. Send a new ray against the plane and get the intersection point
2. Calculate the distance between the first picture intersection point and the second plane intersection point

When knowing the mouse coordinates and the distance to where the picture is located it is easy to set the picture’s new position. Grabbing and moving view is another navigation technique that is based on the same idea as when moving objects but is using grid planes and not pictures as target in the first phase. It basically navigates the user in the opposite direction of the dragging direction.

4.1.2.2 Throwing
To be able to throw a photo then objects must have velocities. Velocities are supported by the framework through transformation nodes. Throwing is also heavily dependent on the select and drag algorithm since the photos are placed on a certain depth. The Photo Viewer application uses the
framework to get and set the photo position. Velocity of the photo is dependent on the mouse and the delta vector of the photo movement is stored as a velocity in a list. The list can store any number of velocity vectors but in this example application it is limited to a few. Throwing is done in three steps, clicking, dragging and dropping. The first click gives a starting point. Dragging gives velocity samples while the user is moving the photo with the mouse cursor. Dropping gives the photo a final velocity which is a median of the collected velocities. The median of the collected values are used to smooth the outcome, otherwise it would be possible to get zero or extremely fast velocity if any jump in movement occurred.

4.2 Scene Viewer

The scene viewer example is more focused on using three dimensions such as several larger models with textures and shader effects. This section also includes how models are constructed and loaded by the framework as well as some screen shots from the original scene in Autodesk 3D Studio Max\textsuperscript{10}.

Figure 4.19 below is a screen shot from the scene viewer. The scene is built of multiple scene graphs that are linked into the uniform grid. The camera in this example is not restricted which makes it possible to fly around and look from every angle of the scene.

Figure 4.19 – Models loaded into the scene viewer.

4.2.1 Models

Most models in the scene viewer are constructed from several meshes, textures and nodes that are linked to a single scene graph, which means that all connected parts will move if the scene graph object moves. It is possible to assign multiple objects the same texture and shader ID ($tID$ and $sID$). Every mesh has its own transformation node which makes it possible to move and rotate them to the wanted position. Below is a list of meshes and materials for the statue in Figure 4.19.

1. Body mesh, dark shiny cloth material.
2. Left and right arm mesh, dark shiny cloth texture.
3. Hat mesh, dark shiny cloth texture.
4. Left and right leg mesh, gray shiny cloth texture.
5. Head mesh, white shiny stone texture.
6. Left and right hand mesh, shiny stone texture.
7. Left and right feet mesh, shiny stone texture.

4.2.2 Modeling

Figure 4.20–4.23 below are four screen shots taken from different angles of the original scene while being modeled in Autodesk 3D Studio Max. Each model of the scene was texture mapped and converted to a mesh of triangles before being saved to the .3ds file format. Note that lights and material effects such as blinn and phong lightning models and their settings are not saved to the .3ds file format, these has to be set manually after being loaded by the framework. All polygons have the form of a triangle in mesh mode, which is a requirement when loading through the framework.

Figure 4.20 – Original scene from top.

Figure 4.21 – Original scene from left.

\textsuperscript{10}Autodesk 3D Studio Max: (usa.autodesk.com/3ds-max) Accessed: 23112011
4.3 Floating Controls

This scene is built up by three frames viewing different pictures and three buttons. The purpose of this example is to show that the framework supports event handling and that textures easily can be swapped through node search and change of texture ID. There are three events which are connected to the geometry nodes of the buttons. Clicking any button except the middle one will loop the pictures viewed by the frames back and forth. The middle button will close the application. Every geometry node can hold user made events that are triggered if the object is hit during a ray intersection test. All triggered events are caught in the event loop and from there it is possible to read information passed from the node. By using a list of texture IDs and it is possible to step forth and back in the list for the desired effect. Below is a small code example:

```c
if(event.zui.user.code==UEGEOMETRYCLICK){
    int id = *(int*)event.zui.user.data1;
    int in = *(int*)event.zui.user.data2;
    SceneGraph* sg = ZUI_GetObject(id);
    if(sg){
        MaterialNode* g = (MaterialNode*)
        sg->GetNodeByName(SGMATERIAL);
        GLint ta = g->texInfo[0].GetTexture();
        (change texture based on id & in here)
        g->texInfo[0].SetTextureID(ta);
    }
}
```

The event that is sent to the event loop can be created by the piece of code below. The first row allocates memory for the new event e while the second is a user defined code to be able to distinguish it from other events in the event loop, the third sets the type to user event which is an unused range in SDL. The next two rows set two integers where the first is a known scene graph id which is supposed to change when the event is received. The second one is the ID of the node in the selected scene graph to be found or changed. The ID has to be assigned to the node manually and in this case it is called SGMATERIAL and has a value of one. The last two rows are pointer assignments which makes it possible to read the values in the event loop.

```c
ZUI_Event* e = new ZUI_Event();
e->zui.user.code = UEGEOMETRYCLICK;
e->zui.type = SDL_USEREVENT;
*sid = 4;  // allocated int*
*inc = SGMATERIAL; // allocated int*
e->zui.user.data1 = (void*) sid;
e->zui.user.data2 = (void*) inc;
```

Below is a short piece of code showing how to assign name or search value of a material node. The first row allocates memory for the new node, the second assigns which texture to use through the integer textureID, it also assigns the texture unit which is the first one and the name of the texture in the shader to pass it to is named texture1. The third row assigns which shader to be used. In this case the loaded shader is located on the third place in the shader container. The last row needs to be set if it should be possible to search for the node by its name or id, which in this case is SGMATERIAL with value of one.

```c
MaterialNode* mn = new MaterialNode();
mn->InitializeTexture
    (textureID, 0, "texture1");
mn->InitializeShader
    (sc->GetShaderObj(3)->shaderProgramID);
mn->SetName(SGMATERIAL);
```
4.4 Texture Blending

This is a small example containing a 3D-model with texture blending (see Figure 4.25–4.28). Texture blending is achieved by assigning three texture IDs on three different texture units to a single material node. It is possible to link the three textures to a shader where the first and second are different materials and the third is texture mask or blending texture. The texture mask is a gray scale image and decides if texture one or texture two should be visible at every fragment. Black means the fragment color of texture one and white means the fragment color of texture two, gray means equal amount of color from both textures.

5 Results

As there still are many performance related algorithms to implement to see the framework as complete it would be wrong and misleading to measure performance at this point. The two major data structures the uniform grid, the scene graph as well as algorithms implemented into the framework are well motivated for their purposes: performance and flexibility, which is what the first section is about.

During this project a small pilot study was made to collect information by evaluating an early prototype of the Photo Viewer, the results are presented in the second section.

5.1 Performance

The choice of data structures and algorithms for the 3D-ZUI framework turned out to work well for the example applications. The scene graphs are flexible and easy to extend if needed due to independent behavior and object oriented design, these were the most important design goals of this project. When many objects are loaded into a 3D-ZUI it is not enough to use a brute force algorithm for picking and collision detection. To improve performance of the 3D-ZUI a three-dimensional uniform grid was implemented into the framework. The theoretical performance improvement comparing to not using a grid at all for picking and collision detection is huge. Exactly how many objects the framework can handle remains to see as it greatly depends on the grid-, voxel- and object size [11], how these structures combined affects performance is not yet measured in numbers and is listed as future work.

When looking closer on theoretical performance for ray traversal the worst case traveling path for a ray through a two-dimensional uniform grid of size 6x6 is from voxel (0, 0) to (5, 5). For a correct and precise path it requires traversing through 11 voxels using the traversal algorithm described in section 3.2.2.1. The best case is 1 voxel if the ray...
starts for example in voxel (0, 0) and is pointing in a negative direction or that voxel (0, 0) is holding the wanted information. A brute force algorithm would always end up passing 36 voxels which is about 3.3 times slower and this is for the two-dimensional algorithm only. Picking without the grid would have to loop through all objects for every pick instead of the ones that are placed nearby the click ray. Different kind of applications may use a much larger grid with varying voxel sizes. The Photo Viewer application grid is set up to about 5000 voxels which is a rather small zoomable user interface that isn’t utilizing the depth very much. The 3D-DDA algorithm by John Amanatides and Andrew Woo [3] is considered a fast and memory conserving algorithm.

The same principle as for ray traversal applies for object/object collision detection. The uniform grid is used for the collision detection to reduce the number of objects tested against each other. A spatial partitioning structure is very effective performance wise when it is encapsulating many objects. The grid is accelerating collision detection by searching the voxels the objects are registered in to quickly find nearby objects. Objects that are not registered in the voxels are not tested for intersections. In an application with 5000 objects there might be only 10 that are colliding at the same time in different places. Since an object otherwise has to be tested against all other objects, it would give about an average of 25 million object/object intersection tests. The grid also needs to relink moving objects in the grid which requires computations. The framework is using AABBs to insert and remove objects from the grid. This method is very fast and does not require any intersection tests but some objects may be registered in more voxels than necessary. It is important to balance the grid size and voxel size for the objects in the application for optimal performance [11]. Too large voxels would increase the number of intersection tests with nearby objects while too small voxels would increase the time spent on relinking the grid, voxel search around objects and also ray traversal in the grid.

The scene graph is structured as a BVH which means that all BVs in the structure don’t need to be tested and by doing so it becomes fast to traverse and test for intersections. The current version of the framework allows BVs to overlap which should be kept in mind when building the scene graphs. The scene graph supports dynamic models at the time and it is possible to move nodes further down the tree. BVs does not automatically rebuild in the current version of the framework so picking will only work correctly on static models or manually updated BVs. This is also one of the reasons why switch nodes are not yet implemented.

It is possible to improve performance even further. Some limitations that were expected and that also appeared during extreme testing were related to memory and rendering power. These issues are known and solutions are available. Two kinds of culling, a texture manager and also a technique called LOD (Level of Detail) are currently being looked into and are listed as future work.

5.2 Pilot Study

An early prototype of the Photo Viewer was evaluated by a small group of persons. The purpose of this pilot study was to get feedback and guidelines by investigating how people felt about the concept and at the time implemented features. The test application had several predefined photos that were loaded when the application was started. There were no restrictions of how to use the application. After the test run the users were asked to fill in an evaluation form containing questions about how they experienced the Photo Viewer and its current features.

Most questions asked were directly related to the Photo Viewer (framework) and features. The scale 1-9 was used so the user could give an accurate answer to each question. A summary of the evaluation is represented by a table (see Table 1). The purpose of the table is to show the average response by using a scale of 1-9 for every question. A score higher than 5 is good and a score lower than 5 is not as good, a value of 5 is neutral.

Table 1 – Summary of evaluation.

<table>
<thead>
<tr>
<th>QN</th>
<th>TYPE</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Age</td>
<td>44</td>
</tr>
<tr>
<td>Q2</td>
<td>Yes/No</td>
<td>3.3</td>
</tr>
<tr>
<td>Q3</td>
<td>Bad/Good</td>
<td>9.0</td>
</tr>
<tr>
<td>Q4</td>
<td>Bad/Good</td>
<td>8.3</td>
</tr>
<tr>
<td>Q5</td>
<td>Bad/Good</td>
<td>8.3</td>
</tr>
<tr>
<td>Q6</td>
<td>No/Yes</td>
<td>8.7</td>
</tr>
<tr>
<td>Q7</td>
<td>Bad/Good</td>
<td>9.0</td>
</tr>
<tr>
<td>Q8</td>
<td>Bad/Good</td>
<td>8.7</td>
</tr>
<tr>
<td>Q9</td>
<td>Bad/Good</td>
<td>8.0</td>
</tr>
<tr>
<td>Q10</td>
<td>Yes/No</td>
<td>7.7</td>
</tr>
<tr>
<td>Q11</td>
<td>No/Yes</td>
<td>7.7</td>
</tr>
<tr>
<td>Q12</td>
<td>No/Yes</td>
<td>8.7</td>
</tr>
<tr>
<td>Q13</td>
<td>No/Yes</td>
<td>5.3</td>
</tr>
<tr>
<td>Q14</td>
<td>No/Yes</td>
<td>9.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.2</td>
</tr>
</tbody>
</table>

The application and evaluation form was handed out to a small group of test users. Below is a quick summary of the results.

Question one (Q1) and Q2 which are age and experience of similar applications were asked to be
able to categorize results and are not part in the final score. Questions Q3 to Q8 are related to the features currently implemented: picking, colliding photos, screen tracking and zoom. Results show averages between 8.3-9.0 which are on the positive side. Q9 asked how the user felt about the application in its current state. All users gave positive feedback of avg 8.0 on this question. Q10 asked if the user felt disoriented at any time and got an average of 7.7 which means very little. Q11 to Q13 was questions about some features to be implemented. Q11 asked if the user would like to have the option to see a small overview window. A feature like this may for example prevent disorientation and improve navigation. All users thought an overview window was a good idea and got an average of 7.7. Q12 asked if the user wanted to see an option to enable free rotation of the photos (for example if a photo was taken upside-down). This idea was appreciated with an average of 8.7. Q13 asked if the user would like to have the option to enable completely free camera movement (free cam). Two users thought the option would be okay while one completely disagreed. The one that disagreed was the test person with most experience of zoomable photo software. Q14 asked if a right-click menu would be available. The test users responded that the idea was greatly appreciated. The test users also had a free text section for comments, ideas, and thoughts. Some answers were:

- Add support for using keyboard arrows for camera steering.
- Notification that a photo has moved to the opposite side.
- Selection of more objects at the time, multi selection (square selection).
- Change of background image and grid color.
- Ability to grab a photo and drag it over other photos, using shift for example. When dropping the selected photo the under laying photos should move away.

Overall the test users liked the the early version of the Photo Viewer, its currently implemented features and some of the ideas in the evaluation form. The total score of the evaluation (Q3-Q14) ended up at an average of 8.2 of a maximum of 9 which gives a positive impression. Many options and features have been implemented since the pilot study, including some ideas listed in the evaluation form. An evaluation of a fully developed Photo Viewer would give interesting and valuable feedback for further development of the 3D-ZUI. A more thoroughly evaluation with a larger group of test users is listed as future work.

### 6 Conclusions & Future Work

There already exists 3D-ZUIs today but they are far from as common as 2D-ZUIs. Most navigation-, photo viewer- and drawing applications such as Google Maps, Live Photo Gallery and Adobe Photoshop among other are built as 2D-ZUIs with a lot of different features that is shaping the application to what it is. Most 3D software are built like 3D-ZUIs, for example creation software like Autodesk 3D Studio Max, Maya and Blender but also map and navigation software as Google Earth and desktop like applications like BumTop.

One of the most similar 3D-ZUIs that exist today is the desktop software BumTop. BumTop aims to be more physically correct in many ways and it has support for pen input and features like piling, lasso selection and throwing [1]. The environment is like a box or walls with objects common on regular desktops in real world. Connected devices are popping up on screen if connected and so on. It is not entirely fair to compare the framework of this project with an application but if we look at it from the ZUI perspective there are a few similarities such as picking, dragging, zooming and navigation. BumTop has collision detection and physics implemented with support from a physics library for a more lifelike feel. When this 3D-ZUI project started it was not convincing that ”close-to-reality” physics was the way to go. Still, it is possible to emulate such physics effects and also to extend the current framework if needed. A goal was to create a framework and sample applications that are easy to use and does not necessarily have to reflect reality in any way. The framework does support collision detection through the uniform grid which seemed necessary so that objects in different size and shape don’t encapsulate other objects. It is possible to disable collision detection entirely or on certain objects if necessary. An example would be if there are many photos (as in the Photo Viewer) on the same plane in space and the photo object in the middle needs to be moved towards the corner, in a case such as this collision detection could temporarily be disabled while for example holding the shift-key while moving the object. The somewhat limited or simplified navigation in BumTop seems to be a good choice considering the possibility of users getting disoriented [18, p1]. It is unknown which data structures and algorithms that were used in BumTop to create this box or room space. The framework is designed to be able to handle large scale interfaces with many objects and to be as flexible as possible. BumTop evaluation was successful and people had positive reactions. Responses were that it felt familiar and didn’t require much training [1].
A prototype version of the Photo Viewer was tested by a small group of people. Getting feedback on the Photo Viewer application was very interesting and many features and options have already been implemented since the evaluation. Once the Photo Viewer is fully developed it would be of great value to perform a larger user study to collect feedback for further development of the 3D-ZUI, this is now listed as future work. Limiting the navigation system felt necessary for the Photo Viewer to avoid users getting disoriented. After some testing and the results from the evaluation of the prototype the free cam was disabled. The view got locked on looking straight forward while being able to move in the same plane as the screen and forward (x, y), with the ability to zoom of-course (z). This also includes limiting the camera so that it isn’t possible to fly around in loops by limiting yaw, pitch and roll parameters. The camera related to the Photo Viewer needs further investigation. Allowing semi movement through multiple layers, real time clipping or cropping, rotation and zooming of textures on the picture plane are some features that might be available in future versions.

Most features related to the uniform grid and scene graph structure was implemented and used in the Photo Viewer such as dragging, grabbing, throwing, rotations, navigation, zooming and collisions which all worked as according to the plans. This example also includes 3D-models, textures and shaders that react to lights placed in the scene. An issue that was expected and also appeared while developing and testing the application was the memory issue. It was not possible to load as many textures or photos as wanted due to insufficient amount of video- and system RAM. There are ways to solve these issues where one of them is a texture manager that decides if a texture has to be loaded into memory or not, this technique would have to work together with a culling algorithm. To improve performance even further it would be possible to implement LOD which is a technique to decrease quality of objects that are further away from the camera while maintaining full quality on objects near the camera.

Uniform grids are commonly seen in ray-tracing [11, 24] and games [25, p249, p385] mainly to reduce intersection tests while the scene graph is widely used to represent 3D-objects due to flexibility and the possibility to use it as a bounding volume hierarchy [21, 23]. The uniform grid and scene graph were chosen and successfully implemented into a 3D-ZUI framework and the two structures are well motivated for accelerating performance in large scale interfaces as well as providing flexibility when setting up and handling 3D-objects. A design goal was to implement the framework in an abstract and object oriented way to easily be able to adapt and extend its capabilities. There are still some performance related algorithms that can be implemented into the framework as some issues were confirmed during early testing. A general technique to improve the performance of a 3D-ZUI but also 2D-ZUls is culling which is used to filter away objects that are not visible and by doing so save computing power, however, performance may vary depending on implementation [14]. Culling has been used together with scene graphs in previous works where the graph is built like a bounding volume hierarchy [21, 23]. There are two kinds of culling needed for the framework. The first which is called frustum culling is comparing object volumes against six frustum planes; left, right, top, bottom, far and near. Everything outside the frustum “box” is not to be drawn on screen. The second method is called occlusion culling which task is filter away objects that are covered by other objects and therefore not necessary to draw.

Future work also involves if the uniform grid can be used for culling (grid culling). The idea is if it is possible to only render objects that are inside the view frustum without the need to loop through all objects in the scene. If thousands or millions of objects are loaded into the scene maybe only a few hundred are visible in front of the camera, saving the computing power required by all other objects may prove to be worth the cost of finding the visible objects in the view frustum. Square selection, a type of multi selection uses the same principle as frustum or grid culling. This feature could be implemented using the uniform grid, it might be useful unless there are very small voxels in the grid. A possible solution to improve search inside the frustum could be to use subgrids or hierarchical grids [11] to minimize grid traveling, it would however then be more costly when relinking objects in the grid. Considerable speedups in culling could be to use simple bounding volumes for every scene graph, for example a sphere. A sphere is very fast to cull and requires only a position and radius. Several fast bounding sphere construction algorithms are available [15]. In addition, slab cut balls could be supported to improve the otherwise quite loose fit of the bounding sphere without introducing too much complexity in the bounding volume representation [16]. There are several kinds of culling and many ways of doing it, which makes it an interesting topic to investigate.

The currently implemented picking method uses bounding volumes to decide if an object is hit or not which makes it fast but the precision will suffer if the BV isn’t tight enough. During this project a common ray/triangle intersection test was implemented for testing purposes but is not yet a part
of the framework. The intersection test is to be added to the picking algorithm and is listed as future work. The picking algorithm is accelerated through the uniform grid using a fast and precise traversal method originally developed for ray tracing [3, p2] and by using a bounding volume hierarchy in the scene graph which have been done several times before [21, 23].

Colliding objects are using the BVs to decide if a collision has occurred. The BV/BV intersection methods used in this project are fast but the precision may vary depending on the shape of the model. A triangle/triangle intersection test is to be investigated and added to the object/object collision detection algorithm for optimal precision.

Adding a light node for the scene graph has been looked into during this project. The idea is that certain object may have their own local lights that are following the transformations of the object, for example a torch or to brighten up a certain interesting area. Such nodes have been implemented and used in scene graphs before [23].

Much like the light node it would to be possible to assign cameras to scene graphs, as in previous work [23]. The cameras can be seen as custom cameras for specific models with their own positions and parameters. For example if having 3D-menus or 3D-text it can be possible to use a predefined camera for the specific purpose, or specific models. An idea is to be able to render camera icons by selecting an option, and by clicking the camera icon of an object the user will transfer to the local camera and so on.

A switch node can have unlimited or limited number of states which while in traversal phase will decide which child to travel to [21, 23]. A node like this could prove beneficial when using animated objects such as buttons. For example a click changes the state of a button and pushes it forwards or backwards, switching between several models as in level of detail as examples. This node type requires that the bounding volumes to be automatically rebuilt which is not yet supported.

Further investigation needs to be done regarding material restoration. The current solution is using get, push and pop stack calls using the OpenGL API which can cause quite a performance impact. A possible solution might be to let the scene graph store the information and thus reducing API calls.

The old 3ds file format is lacking some important information about the model such as normals, material properties and light information. The normals are calculated by the framework which can cause some differences between the original model and the one loaded into the ZUI if it is not using normal mapping. Light information such as position, type and various color properties are missing and have to be set up manually by using OpenGL standard functions. Another solution for the light information might be to write a plug-in to 3D Studio Max and thus be able to store and load light information separately.

Touch optimized Graphical User Interface (GUI) and navigation is now listed as future work as there are many fast tablets and hand held computers available in the market. Utilizing multi touch might prove useful when dealing with multiple dimensions as it now require both limited input devices as mouse and keyboard and nested two-dimensional menus to express the functionality. When using multi touch it would be possible to select more objects at the time with different fingers while the other hand rotates or zooms the view by using different gestures. It would possibly feel more real using the hands instead of a mouse cursor and interaction would become easier to understand and use. Adding multi touch capabilities to this project feels at this point necessary and very interesting as todays input devices are somewhat limited.

Some parts of the framework are still limited to two dimensions, such as menus and text. The framework can handle objects that are linked to events which make it possible to build 3D-controls as in the Floating Controls example. It is therefore also possible to extend the framework to support 3D-menus. Text can be rendered onto a texture and rendered in 2D-mode on screen or put on a 3D-object. To be able to draw 3D-text would extend the flexibility of the 3D-ZUI and it would be possible to make proper use of shaders for lightning and surface effects.

The 3D-ZUI framework is a work in progress and many improvements and features are listed as future work. There are still explorations in what areas 3D-ZUls could be used in and how to give them a fair evaluation. This project was formed out of curiosity of how 3D-ZUls can be built and what they can be used for. Some applications are already build as 3D-ZUls, for example creation or modeling applications such as 3D Studio Max, Maya and Blender but also the desktop alternative BumTop and map navigation software Google Earth that has a surface of satellite images, street views and 3D-buildings. Some scene graphs, toolkits and frameworks for building interactive 3D applications have been developed before and scene graphs seems to be a common factor [14, 21, 23]. Many other applications but also game engines are using algorithms and data structures similar to the ones brought up in this project. A reason for using 3D-ZUls could be the look and feel, a more known and life like view as well as interaction with 3D-objects, interfaces merging into the common life. It is possible to develop desktop interfaces and worlds with sup-
port from this framework but it is mainly developed to be as flexible as possible so that developers can choose what to build upon it. If looking at near future it would be interesting to see how a 3D-ZUI would do as an interface of an operating system as well as a base for games. The framework is still a work in progress and further investigation and implementation is necessary to see it as complete package.
7 References


