Faculty of Economic Sciences, Communication and IT
Department of Computer Science

Christoffer Bengtsson
Roger Hemström

Warehouse3D

A graphical data visualization tool

Computer Science
C-dissertation (15 hp)

Date/Term:	2011-01-20
Supervisor:	Kerstin Andersson
Examiner:	Donald F. Ross
Serial Number:	C2011:01

Warehouse3D:
A graphical data visualization tool

Christoffer Bengtsson, Roger Hemström

This report is submitted in partial fulfillment of the requirements for the Bachelor’s degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Christoffer Bengtsson

Roger Hemström

Approved, 2011-01-20

Advisor: Kerstin Andersson

Examiner: Donald F. Ross

Abstract

Automated warehouses are frequently used within the industry. SQL databases are often used for storing various kinds of information about stored items, including their physical positions in the warehouse with respect to X, Y and Z positions. Benefits of this includes savings in working time, optimization of storage capability and – most of all – increased employee safety.

IT services company Sogeti’s office in Karlstad has been looking into a project on behalf of one of their customers to implement this kind of automated warehouse. In the pilot study of this project, ideas of a three-dimensional graphic visualization of the warehouse and its stored contents have come up. This kind of tool would give a warehouse operator a clear overview of what is currently in store, as well as quick access to various pieces of information about each and every item in store. Also, in a wider perspective, other types of warehouses and storage areas could benefit from this kind of tool.

During the course of this project, a graphical visualization tool for this purpose was developed, resulting in a product that met a significant part of the initial requirements.

Acknowledgements

We would like to thank Thomas Heder at Sogeti Karlstad for his support and help during the course of this project.

Also, a big thank you goes to Bengt Löwenhamn and Åsa Maspers at Sogeti Karlstad for inviting us to do this dissertation work.

Last, but definitely not least, we would like to thank our advisor Kerstin Andersson at Karlstad University for her great support and feedback.

Contents

List of Figures

List of Tables

1 Introduction

Sogeti is a consultancy specializing in local professional IT services [1]. Keeping industrial IT as one of their main branches, their local Karlstad office is looking into a project on behalf of a customer with the goal to get one of their warehouses fully automated. Within that project, an idea of a tool for graphical three-dimensional visualization of a warehouse has come up. This kind of tool would serve two purposes: first, provided that it is reasonably expandable, it could be useful in many different projects that manage warehouse or storage data. Second, being a quite fancy graphical product, it could probably be used by Sogeti for demonstration and marketing purposes in customer meetings.

The assignment for this dissertation project was to create a tool that would meet these criteria. This tool should be able to display a warehouse and its stored items from several different three-dimensional perspectives. By clicking these items, the user would then get different kinds of information about them, such as price, date of delivery or something else of interest for the particular warehouse.

1.1 Technical Project Requirements

The main goal of this project has been to create a graphical user control (further on called 'Warehouse3D', or simply 'the product'), showing a 3D view of a warehouse and its contents. The following sections describe the requirements for the project, divided into three priority categories.

1.1.1 Essential Requirements

At a very minimum, the following functionality should be implemented:

- Display a warehouse and its contents from two different angles; a centered bird's eye view from straight up above, and a three-dimensional perspective view. Figure 1.1 shows the sketches of these ideas.
- Adding and removing a coil at a specified position in the warehouse.

Moreover, an application for demonstration and testing purposes will be developed.

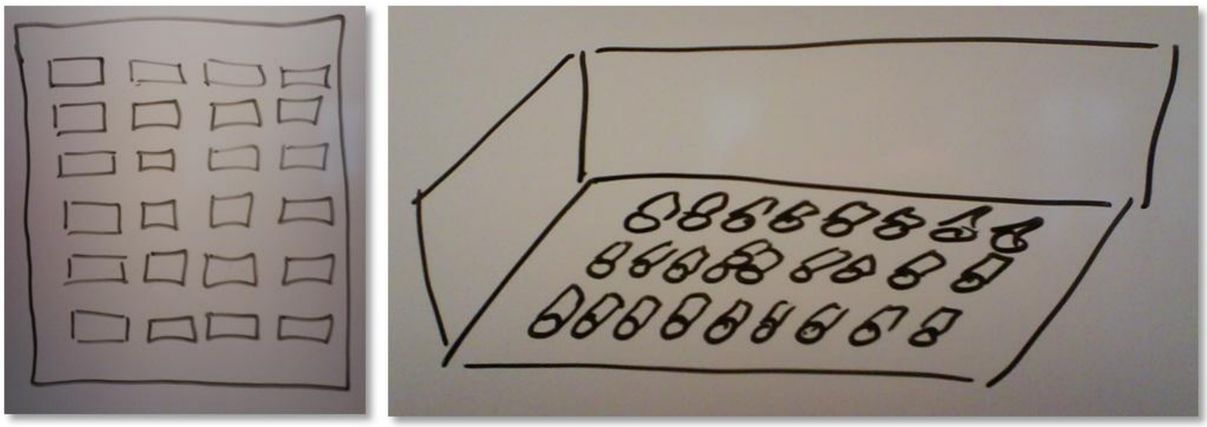


Figure 1.1: Sketches of a warehouse seen from two different angles

1.1.2 Important Requirements

In addition to the above requirements it is also important, but not crucial, that the following features are included in the product:

- The design should be open for extensions and further development such as using other geometric shapes than cylinders as warehouse objects.
- Showing and hiding warehouse objects in the view.
- Coil slots should be displayed on the warehouse floor, preferably as squares or something similar.
- A user should be able to select (click) one or more warehouse objects to get detailed information about these.
- Three-dimensional camera movement: the user should be able to freely “float around” the model to be able to view it from every conceivable perspective and from virtually any distance at all – even close up, just like standing only a few centimeters from a coil.

1.1.3 Further Development

If all the above requirements should be done sooner than expected, there is also some additional functionality to look into:

- Displaying non-rectangular warehouses (for example L-shaped or T-shaped).
- Show the ID of a warehouse object as a glued-on label on the object.

1.2 Chapter Overview

The process of developing the 3D graphical tool is discussed throughout this treatise. Chapter 2 keeps the focus on the background and project ideas along with the goals, purposes and some overview of the technical tools used in the project.

Some general information about 3D computer graphics can be found in Chapter 3 as well as some overview of how this technique is treated in Windows Presentation Foundation.

The actual work and implementation is brought up in Chapter 4. Initially, a first prototype is presented. It was developed with the purpose to get a feeling for the project. The chapter also covers the development process of the final product.

Chapter 5 highlights the results of this project. The product is evaluated and the functionality is compared with the initial requirements. The problems encountered during the course of the project are also brought up.

Finally, Chapter 6 presents some ideas on extended functionality and further development for future use. Experiences gained from working with the project are also summarized and presented.

2 Background

This chapter introduces the actual warehouse associated with the technical product of this dissertation project, along with some general information about how it is currently managed. Furthermore, the chapter also contains the technical ideas behind the Warehouse3D tool and its associated application for demonstration purposes, which is also a part of the dissertation project. This is done by explaining the relationship between the product and its demonstration application, and the difference of usage between them both.

2.1 Introduction

Information technology consult company Sogeti is currently looking into a project on behalf of a customer within the steel producing business, with the goal to get their manually controlled warehouse fully automated. The main reason for this is employee safety; the huge steel coils in store weigh several tons and can cause severe damage in an accident. In addition, Sogeti has also found a number of other benefits of automation, including savings in working time, increased storage capability and a minimization of cassation.

For the purpose of getting a clear overview of the warehouse, Sogeti has come up with an idea to develop a tool used to display a graphical three-dimensional visualization of the warehouse and its coils. This tool will give the user a good insight into which coils are physically in the warehouse, as well as data about each coil such as location, width, diameter and more.

Since the automated warehouse for security reasons would be closed and fenced, the operators have a very limited insight in what it actually looks like. Both simple questions like “where is the coil X located” as well as more complicated issues like “the database says there should be forty-eight coils, but I can only see forty-five in the warehouse - where are the missing coils” can easily be handled and solved with a graphical overview at hand. Warehouse3D is the tool that serves this purpose, and will be presented in this dissertation.

2.2 The Warehouse

The warehouse consists of a number of rows of wedges holding huge steel coils. A crane is used to move the coils in and out of the warehouse, as well as changing places between them when needed. Currently, this crane is controlled with a hand-held device by an operator. In order to deliver a coil out of the warehouse, the operator first needs to manually locate it and then steer the crane to the correct position before grabbing and lifting it.

2.3 The Product

This section will describe the actual product to be developed and implemented during the course of this project.

2.3.1 Product Concept

The purpose of the product is to visualize the data held in the warehouse database as a three-dimensional view of the actual storage area. This would give the user a very quick and clear overview of the data, in comparison to displaying it as a traditional database table which often tends to become somewhat cluttered.

So, why 3D? The main reason for this is the fact that two dimensions would not give the complete picture. If a warehouse item is hidden by another (i.e. standing behind or below), a two-dimensional view would not display this in a proper way. A solution for this could be to create several 2D views - one from above, one from the right, one from the left, and so on. However, one single 3D view would most likely give the user a better overview than several different 2D ones, especially if there is also an option to move the "camera" back and forth in the view.

2.3.2 Technical Design

Since the final product will not be an application in itself but merely a graphical visualization of some portion of underlying data, it will be implemented as a reusable control (discussed more in Sections 2.3.3 and 4.2.1). In practice this means that it will be designed to be consumed by other graphical applications in a .NET environment (see Section 2.4.1 for more information about .NET). This also means that, in order to be as versatile as possible for further development, a well-structured Application Programming Interface (API) must be implemented. Therefore, in practice, the product can be compared to any graphical user control, such as a button or a graphical list.

2.3.3 Graphical User Interface

Developing an application with a 3D graphical user interface means walking a thin line between simplicity and advanced functionality, trying to get as much as possible from both concepts. The end user of the application is not necessarily comfortable with navigating in a 3D computer environment - still, it is virtually inevitable to use camera positioning and movement in order to make the application usable. In this case, however, all input controls will be located in the consumer application and not within the user control. Most functionality will be accessible via the public methods of the products API, which means that the consuming application has to implement some graphical controls - buttons, switches, textboxes, etc - and make them call these methods on user interaction in order to use these functions. However, some parts of the functionality will be “locked” within the user control and cannot be altered from outside, such as mouse-click-selection of items (coils) in the warehouse view.

All in all, this means that a developer using the Warehouse3D control in an application will be free to choose which features to use and which to skip.

2.3.4 Test Drive Consumer Application

For testing and demonstrating the product, a consumer application (further on called the demo application) will be developed. This will contain graphical controls for adding items to the warehouse, as well as view-controlling functionality such as changing of camera positions and hiding or showing visual objects. Figure 2.1 shows a sketch of the product and the demo application.

When used within a “real” system, the Warehouse3D control should preferably be implemented to get its values from an underlying data source, such as an SQL Server database, rather than manually from user input. In practice, the outcome of this will be that the Warehouse3D reflects the contents of the data source and adds a graphical item whenever a new object is added to the data source. However, for the sake of simplicity for demonstration, the user will be able to manually add an object right into the graphical view without the need of an external data source.

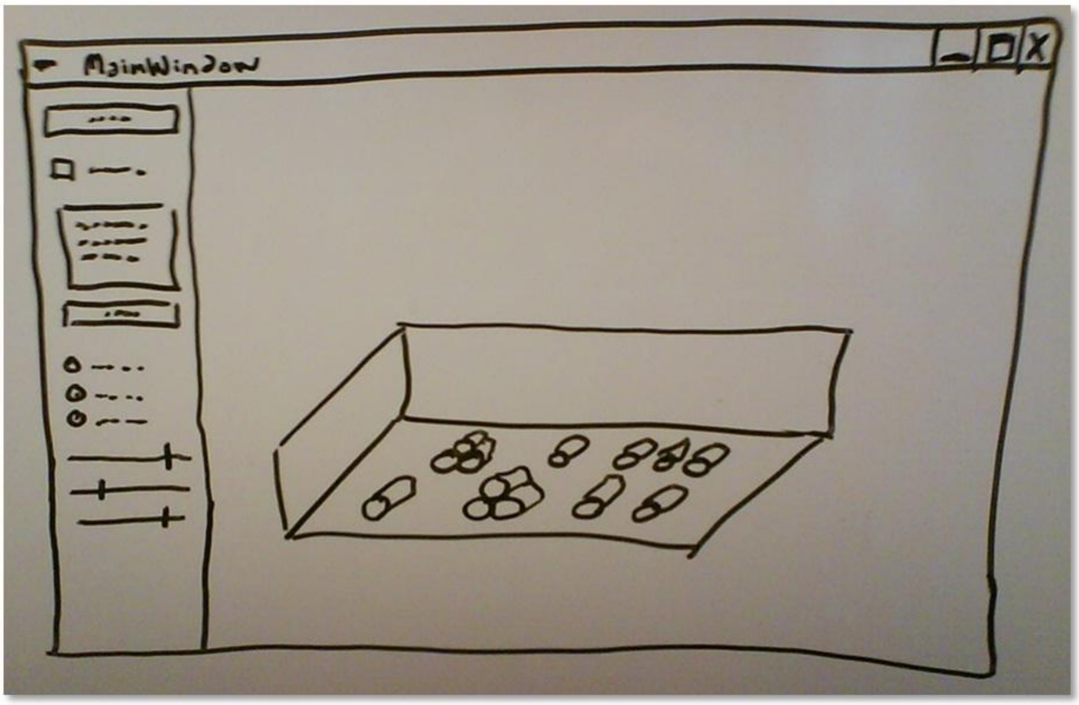


Figure 2.1: Early sketch of the Warehouse3D control embedded in a demo application

2.3.5 Versatility

It is in the nature of a user control to be as all-round as possible. It should not be tied up to only one single application, but rather be reusable in many different environments. This is something to carefully consider during the implementation, in order to keep the code open for extensions and possible further development.

2.4 Tools

Most of the new projects initialized by Sogeti are based on Microsoft products. Since there is no support for 3D graphics rendering in the traditional Windows Forms classes embedded within the .NET framework, Microsoft Windows Presentation Foundation (WPF) will be used, as requested by Sogeti. This will make sure that the product will be fully compatible with other systems developed in a .NET-environment.

This section will give an overview of these concepts along with other associated tools and programming languages used for developing a 3D graphics application for Microsoft Windows.

2.4.1 Microsoft .NET Framework

Released by Microsoft, the .NET Framework is a Windows component that supports developing and running applications written specifically for the framework. It includes a large class library providing features including user interface, data access, file handling, threading, database connectivity, cryptography and networking.

Programs written for the .NET framework execute in a runtime environment known as the Common Language Runtime (CLR), which is a core component of the .NET Framework. Developers using the CLR write code in a language such as C#.NET or Visual Basic .NET. At compile time, a .NET compiler converts the code into a form of bytecode known as Common Intermediate Language (CIL). At runtime, the CLR's just-in-time compiler converts the CIL code into code native to the operating system [2] [3] [4].

2.4.2 Visual Studio 2010

Visual Studio 2010 is Microsoft's latest Integrated Development Environment (IDE) release for virtually all kinds of software development, including console applications, Windows Forms applications, services and web sites for all platforms supported by Microsoft Windows and the .NET Framework. Several different programming languages are built in, such as C/C++, Visual Basic .NET, C#.NET and F#, and support for additional languages is available via separately installed language services. In addition, markup and script languages including Extensible Markup Language (XML), Hypertext Markup Language (HTML), JavaScript and Cascading Style Sheets (CSS) are supported.

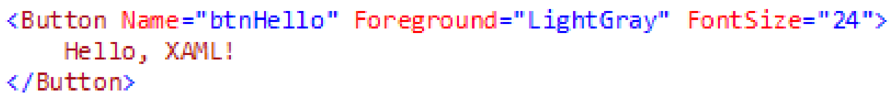
Along with the code editor, which can be found in any IDE, Visual Studio also provides several different graphical editors. There is a Windows Forms Designer for building Graphical User Interface (GUI) applications by dragging and dropping controls onto a form surface, a Class Designer for creating or generating Unified Modeling Language (UML) diagrams, and a Data Designer to graphically edit database schemas, to mention a few. There is also a WPF Designer, which is explained in more detail in the next section (see also Figure 2.4).

2.4.3 Windows Presentation Foundation

Released as a part of the Microsoft .NET Framework 3.0 in 2006, the Windows Presentation Foundation (WPF) is a graphical subsystem for rendering user interfaces in Windows-based applications. It aims to unify a number of common user interface elements, such as traditional forms and controls, fixed and adaptive documents, images, video, audio and 2D/3D graphics rendering [5].

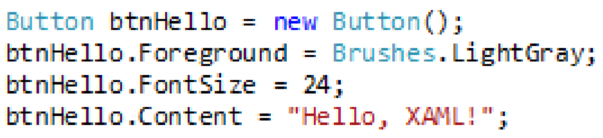
WPF includes an XML-based markup language called the Extensible Application Markup Language (or XAML, pronounced “zammel”). It is primarily used to define user interface elements and data binding, and in some cases it is possible to write entire programs in XAML exclusively. Generally, however, applications are built from both “traditional” CLR compliant code (such as C#) and XAML markup, providing a clear separation between the user interface and the business logic. Figure 2.2 shows a short snippet of XAML. Just like in XML, the three lines comprise a single XAML element; a start tag, an end tag and some content between these two tags. In this case, the element is of type “Button”. The start tag includes two attribute specifications with attribute names of “Foreground” and “FontSize”. These are assigned attribute values, which – also just like XML – requires to be enclosed in single or double quotation marks. Between the start tag and end tag is the element content, which in this case is just the string “Hello, XAML!”

Because XAML is designed mostly for object creation and initialization, the snippet shown in Figure 2.2 corresponds to the C# code in Figure 2.3. As shown, XAML often tends to be more concise than the equivalent procedural code - for example, in the latter, the LightGray value requires to be explicitly identified as a member of the .NET Brushes class [6].



```
<Button Name="btnHello" Foreground="LightGray" FontSize="24">
    Hello, XAML!
</Button>
```

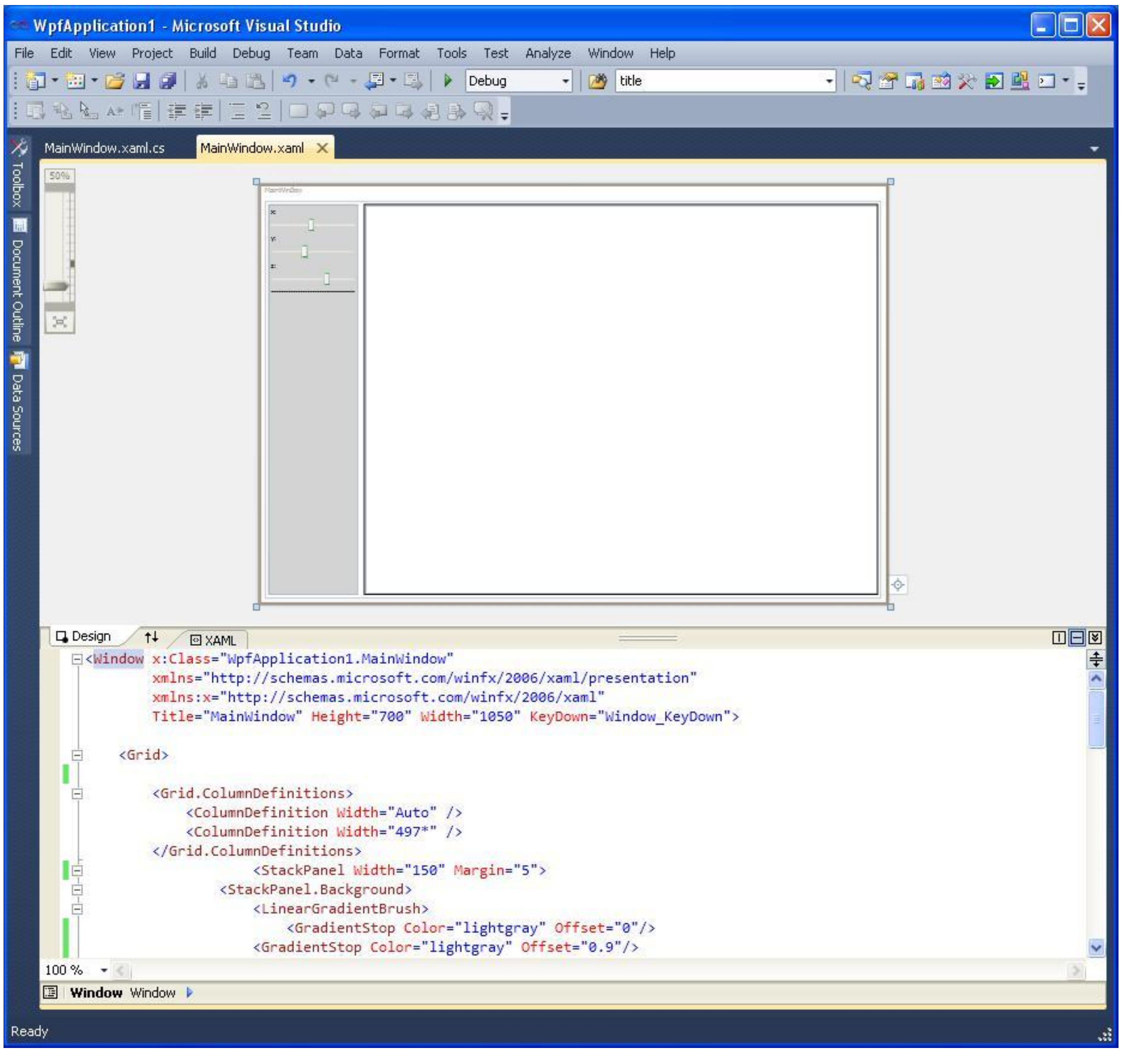
Figure 2.2: XAML code example



```
Button btnHello = new Button();
btnHello.Foreground = Brushes.LightGray;
btnHello.FontSize = 24;
btnHello.Content = "Hello, XAML!";
```

Figure 2.3: C# code example

There is also a distinct separation between the XAML markup and the CLR compliant code in Visual Studio. Like the Windows Forms Designer mentioned in the previous section, the WPF Designer supports drag and drop functionality for adding visual controls to a graphical user interface. Whenever a control is added to the graphical surface, XAML code is generated for that particular control. Of course, it is possible to go the other way around by writing XAML code and see the results dynamically in the designer. Figure 2.4 shows the WPF Designer view in Visual Studio.


WpfApplication1 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Format Tools Test Analyze Window Help
Debug
title
Toolbox
Document Outline
Data Sources
MainWindow.xaml.cs
MainWindow.xaml
50%
Design
XAML
100 %
Window Window
Ready

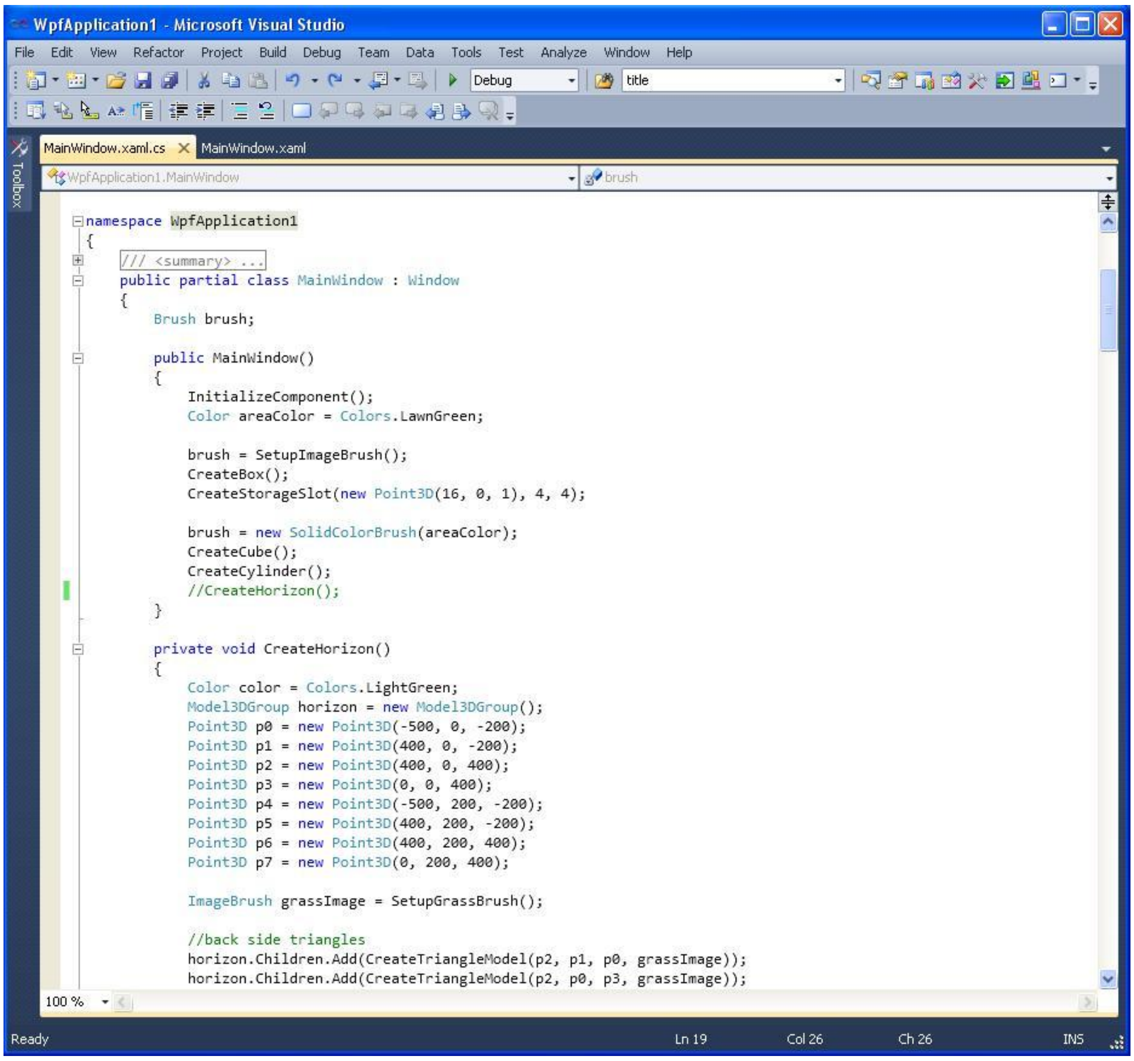
```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="700" Width="1050" KeyDown="Window_KeyDown">

    <Grid>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="497*" />
        </Grid.ColumnDefinitions>
                    <StackPanel Width="150" Margin="5">
                <StackPanel.Background>
                    <LinearGradientBrush>
                        <GradientStop Color="lightgray" Offset="0"/>
                    <GradientStop Color="lightgray" Offset="0.9"/>
```

Figure 2.4: The WPF Designer in Visual Studio 2010

To work with the CLR compliant code - in this case C#.NET - the traditional Visual Studio code editor is used. Simply by double-clicking a file, its associated designer or editor opens up. Figure 2.5 shows the C# code associated with the XAML markup in Figure 2.4 when opened with the code editor.


WpfApplication1 - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Team Data Tools Test Analyze Window Help
Debug
title
Toolbox
MainWindow.xaml.cs
MainWindow.xaml
WpfApplication1.MainWindow
brush
100 %
Ready
Ln 19
Col 26
Ch 26
INS

```
namespace WpfApplication1
{
    /// <summary> ...
    public partial class MainWindow : Window
    {
        Brush brush;

        public MainWindow()
        {
            InitializeComponent();
            Color areaColor = Colors.LawnGreen;

            brush = SetupImageBrush();
            CreateBox();
            CreateStorageSlot(new Point3D(16, 0, 1), 4, 4);

            brush = new SolidColorBrush(areaColor);
            CreateCube();
            CreateCylinder();
            //CreateHorizon();
        }

        private void CreateHorizon()
        {
            Color color = Colors.LightGreen;
            Model3DGroup horizon = new Model3DGroup();
            Point3D p0 = new Point3D(-500, 0, -200);
            Point3D p1 = new Point3D(400, 0, -200);
            Point3D p2 = new Point3D(400, 0, 400);
            Point3D p3 = new Point3D(0, 0, 400);
            Point3D p4 = new Point3D(-500, 200, -200);
            Point3D p5 = new Point3D(400, 200, -200);
            Point3D p6 = new Point3D(400, 200, 400);
            Point3D p7 = new Point3D(0, 200, 400);

            ImageBrush grassImage = SetupGrassBrush();

            //back side triangles
            horizon.Children.Add(CreateTriangleModel(p2, p1, p0, grassImage));
            horizon.Children.Add(CreateTriangleModel(p2, p0, p3, grassImage));
```

Figure 2.5: The Visual Studio 2010 code editor

Concerning 3D graphics, WPF’s primary goal is to bring 3D into interactive user interfaces. It is intended to support a huge variety of hardware platforms by providing a model not based on reality, but with an approximation of reality that is sufficient to allow developers to create visually accepted 3D scenes that can be rendered in real time. The approximations concern particularly the light interaction with objects, and it is discussed in greater detail throughout Section 3.2.4 [7].

2.4.4 Comparing Windows Forms and WPF

The primary goal of WPF is to help developers create attractive and effective user interfaces. Opinions are divided on whether WPF will replace traditional Windows Forms or work as a complement. In graphics-heavy applications displaying animations, 3D graphics or videos, WPF is without any doubts outstanding in comparison. Taking advantage of modern graphics cards, it exploits whatever graphics processing unit (GPU) is available on the system by offloading as much work as possible to it.

Windows Forms, on the other hand, uses GDI+ for graphics. This is a core operating system component that provides two-dimensional vector graphics, imaging and typography. Thus, it can be used for drawing primitives (such as lines, curves and figures), rendering fonts and handling palettes. However, GDI+ cannot animate properly and also lacks 3D rasterization [5] [8] [9].

Moreover, Microsoft says that "Since the initial release of the .NET Framework, many applications have been created using Windows Forms. Even with the arrival of WPF, some applications will continue to use Windows Forms. For example, anything that must run on systems where WPF is not available, such as older versions of Windows, will most likely choose Windows Forms for its user interface. New applications might also choose Windows Forms over WPF for other reasons, such as the broad set of controls available for Windows Forms." [5]. Also, since the Windows Forms technology is older, there is a greater support for this (discussion boards, books, courses, third-party software etc) than for WPF.

2.5 Summary

In this chapter the background and purpose of this dissertation project has been discussed. The goal is to develop a user control for displaying a three-dimensional graphical overview of an automated warehouse.

Developing a user control instead of an application makes the functionality reusable in many different scenarios, but impossible to run and demonstrate in itself. Because of this, an application for testing and demonstrating the product will also be developed as a part of the project. Its purpose is to show the functions of the product and how it might look and act when used in a "real" environment. Along with this, some comparisons have been made between using the product in the demo application and in a larger database-driven system environment.

A brief overview of the warehouse in question has also been given. Some of its current problems have been brought up and the user benefits of graphically displaying the warehouse have been explained.

Some information about the development tools and environment in question - Visual Studio, and Windows Presentation Foundation (WPF) - has also been presented, as well as a brief introduction to XAML, the markup language for creation and initialization of GUI objects.

3 3D Computer Graphics

This chapter will give an introduction to 3D computer graphics and some concepts about the topic in general. Furthermore, it will briefly explain how to work with 3D graphics in WPF specifically, and also bring up some of the built-in classes used for this purpose. Due to the nature of the final product of this project work, the focus has been on 3D graphics programming and rendering, and not on logic and data management. Calculations and computations have therefore mainly included vectors, points and surfaces in 3D space.

3.1 3D Computer Graphics in General

This section will cover some history and concepts of three-dimensional computer graphics in general. The bits and pieces of a 3D object will be discussed along with aspects such as perspective and shading. Some basics in displaying of a 3D scene with the help of a camera and light sources will also be presented.

3.1.1 Introduction

The three dimensions in the concept "three-dimensional" are width, height and depth. Everything we see is three-dimensional - the tree, the car, the computer and so on. 3D graphics sounds like something that would be three-dimensional, but actually it is *not*. The term "3D graphics" is not completely accurate [10]. 3D graphics should actually be referred to as "two-dimensional representations of three-dimensional objects". The objects that a display shows can only be seen the way they are shown - no matter how you move your head around there is no way to see the objects from another angle. Any method to depict a three-dimensional object into a two-dimensional surface is known as a *projection* [11]. Projection is not a new concept that has developed with computer screens or televisions. People have always wanted to depict three-dimensional real world objects onto two-dimensional surfaces. Our ancestors decorated their walls by carving images, and today our magazines, photo albums and so on, are filled with these two-dimensional representations of three-dimensional objects. We are so used to these images, that we can easily perceive three-dimensional representations out of really simple illustrations. Figure 3.1 shows this tendency. There is no doubt to our eyes that the figure represents a cube. In this particular case, however, we cannot tell which is the back side and which is the front side, but we still perceive it as a three-dimensional cube.

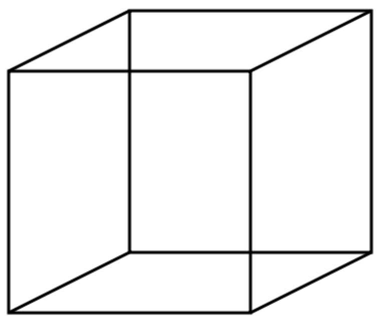


Figure 3.1: A transparent cube

When it comes to computer graphics, the primary motivation for development has been the hardware evolution, along with the availability of new devices [12]. As image-producing hardware entered the scene, software was rapidly developed to use this hardware. Displays were developed that made it possible to display shaded three-dimensional objects. This was an important stepping stone. By calculating the interaction between three-dimensional objects and a light source, the effect could then be projected into a two-dimensional space and be displayed. Such shaded imagery is the foundation of modern computer graphics [12]. Figure 3.2 shows the effect and importance of shading - it is easy to immediately determine the sides of the right cube, while it is impossible to grasp the shape of the left one.

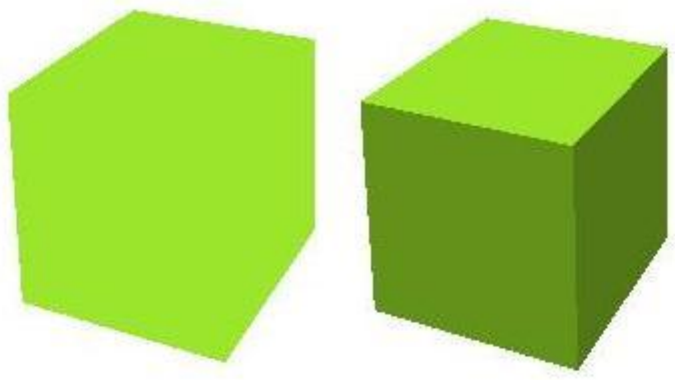


Figure 3.2: Box without shading vs. box with shading

3.1.2 The 3D Space

The objects in computer graphics exist only in the memory of the computer. They are placed in a 3D space, which basically is a mathematically defined cube of cyberspace inside the computer's memory [10]. Cyberspace differs from real world space, as it is a mathematical space that exists only inside the computer. To keep track of the positions of objects in this mathematical space, there is a need for some kind of Global Positioning System (GPS). Coordinates are used for this purpose. Coordinates make it possible to address points due to the width, height and depth of the 3D space. These three values combined make up the coordinates of the point. Coordinates in a Cartesian coordinate system are typically denoted as X, Y and Z.

3.1.3 Building Blocks

In real life, everything is built upon atoms - they are the ultimate building blocks. But if we take it to a higher level of abstraction we could argue that for example a sweater is made upon the use of many threads. Thus, threads can be seen as building blocks for making a sweater. In the same manner there is a need to declare building blocks for graphical three-dimensional figures, in order to be able to create and visualize desired figures and environments.

Three-dimensional figures are traditionally in 3D computer graphics defined by a *polygon mesh*. A polygon mesh is a collection of points, edges and surfaces that together defines the shape of an object in 3D computer graphics [13]. An object consisting of mostly flat surfaces generally only need a small number of points whereas curved and more complex surfaces require a large number of points to approximate the object. This is because curved surfaces actually are just collections of dense flat surfaces. Figure 3.3 shows an example of a 3D polygon mesh. It is quite complicated and consists of several flat surfaces. The individual flat surfaces can be seen as the building blocks for objects in three-dimensional computer graphics, since the surfaces together form the visible object.

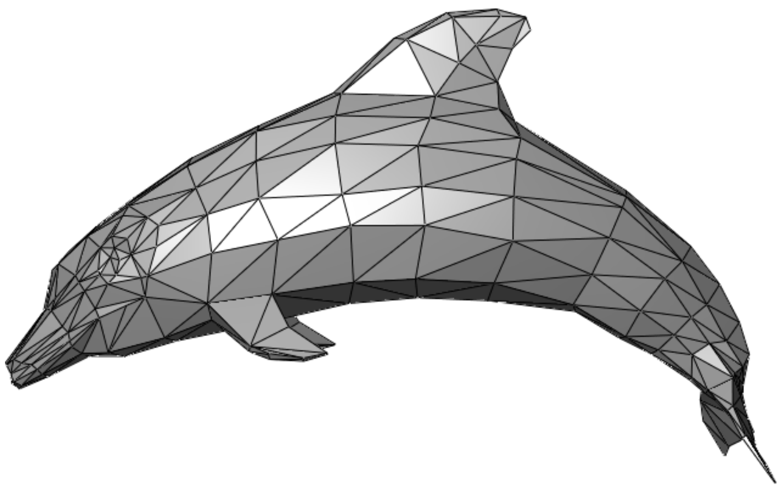


Figure 3.3 : A 3D polygon mesh

3.1.4 Camera and Perspective

To be able to view 3D scenes there is a need for some kind of camera. What you see depend on the position of the camera, in which direction the camera is pointing, if the camera is tilted or not, and the focal length[1].

When projecting three-dimensional scenes into a two-dimensional surface (such as a computer screen), there is a need to consider how to take care of the depth in the scene. There are different approaches to this. Figure 3.4 highlights the differences between an *orthographic* projection and a *perspective* projection. When using a perspective projection, the front parts of a scene or a figure will appear bigger, whilst the back parts will appear smaller. In contrary, an orthographic projection will not show any size difference due to the depth of the particular figure. This can easily be interpreted as if the rear cubes are bigger than the front cube, because that is how humans are used to perceive physical objects. Thus, for most purposes the perspective camera is the most advantageous. One area, however, where orthographic projections are frequently used is in technical drawings where the size measurements of a figure need to be perfectly clear to a viewer [11].

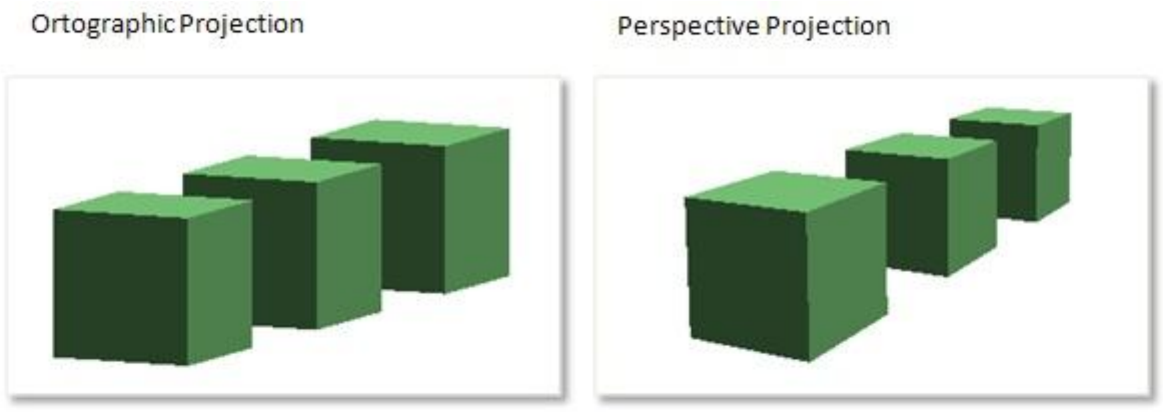


Figure 3.4: Orthographic projection versus perspective projection

3.1.5 Light Sources and Shading

To be able to display photo-realistic scenes, the calculation of light-object-interaction is important. This splits into two fields; local reflection models and global reflection models. Local reflection models consider the interaction between an object and the light source only, as if they were the only ones to exist. In other words, only the reflection of light from the

[1] "The focal length of a lens determines its angle of view, and thus also how much the subject will be magnified for a given photographic position" (McHugh) [14].

object itself is considered. This is realized using a technique called *shading* (see Figure 3.2). Shading can be described as “a process used in drawing for depicting levels of darkness on paper by applying media more densely or with a darker shade for darker areas, and less densely or with a lighter shade for lighter areas” [15]. To give a realistic impression, the shading of the objects has to be considered in relation to several different aspects including the direction and intensity of the light, the position of the viewer and the material of the object. Shades should not be confused with shadows. The difference between these two is that shading is how the object itself reacts to the light, while a shadow is rather an area where direct light from a light source cannot reach due to obstruction by an object.

Global reflection models consider the reflections of light from objects, travelling to other objects in the scene. Thus, the light reflected from a particular surface may have arisen directly from a light source (local reflection model) and/or from indirect light that was initially reflected by another object and was surpassed to the particular surface (global reflection model).

There are different kinds of light used in 3D computer graphics. One kind of light is often referred to as ambient light, and mimics daylight a day when you cannot see the sun. The light seems to be evenly spread without an obvious source of light. Other kinds of light have obvious directions of the light, including distant light, omnidirectional light and spot light. Distant light mimics daylight a day when you can see the sun. The light source seems to be very distant and the light strikes the whole area at a uniform angle. Omnidirectional light imitates a light bulb and emits light in all directions. Spot light can be thought of as a flashlight. The rays of the light are sent in different directions, but with a limited width, similar to the shape of a cone. There are also other variants, but the mentioned ones are the most common in 3D modeling tools [10] [11].

3.2 Working with 3D Graphics in WPF

For many years, developers have used multimedia APIs like DirectX and OpenGL to build three-dimensional graphical interfaces for their applications. However, this difficult and time-consuming programming model and the substantial hardware requirements have kept 3D programming out of most mainstream consumer applications and business software [16].

WPF might hold a solution for this issue. This technology includes lots of classes and structures for building complex 3D scenes which most computers can display without the need of the latest graphics card [16]. This section will give a brief overview of how 3D graphics is treated in WPF.

3.2.1 The 3D Space and 3D Figures

The entire 3D scene is in WPF generally defined inside a *Viewport3D* element, which is a two-dimensional visual element. Viewport3D acts like a window into a three-dimensional scene, and can be used as a part of a larger layout of GUI elements along with panels, buttons, textboxes and so on.

A Cartesian three-dimensional coordinate system is used in WPF, where the “width” axis (X) runs horizontally and increases to the right, the “height” axis (Y) runs vertically and increases upwards, and the “depth” axis (Z) travels from the back to the front of cyberspace. It is important to consider that this is a fundamental coordinate system of the 3D space, and that it is fixed. Depending on the position of the viewer, the relative X axis may not be the same as the fixed X axis for the 3D space. The relative axis and the fixed axis will only be parallel from a given perspective. The fixed coordinate system of the 3D space can be referred to as the *world coordinate system* [10].

Units in WPF are entirely relative. They are not pixels, centimeters or inches - the size of the numbers does not matter other than their relation to the numbers used by other points, and their relation to the position of the camera [11]. This can be compared to a picture or a movie on a screen - it is impossible for a viewer to surely determine the actual size of any object projected on the screen in any other way than compared to another object on the same screen.

Coordinates are used to represent locations of points in the three-dimensional space. To store a location, WPF uses a structure named *Point3D* which stores the coordinates for a given point (that is the X, Y and Z values for that location). A single point in 3D space does not give much of a value though. It is most often preferred to store a collection of Point3D objects, to be able to define the corner points (sometimes referred to as indices or vertices) for a polygon mesh (as described in Section 3.1.3). There is a *Point3DCollection* class that serves this purpose. An object of Point3DCollection, together with information about edges connecting the points (and thus specifying the surfaces) gives a polygon mesh. A class named *MeshGeometry3D* is used for specifying meshes in WPF, and Table 3.1 shows the two essential properties. They contain the information that defines the actual shape of the 3D object. As an example, Figure 3.5 shows three different meshes; all based on the exact same set of points, but with different sets of edges.

Property	**Data Type**	**Description**
Positions	*Point3DCollection*	Contains the locations of the corner points of a figure.
TriangleIndices	*Int32Collection*	Describes how the corner points are connected to form triangles.

Table 3.1: Essential properties of MeshGeometry3D

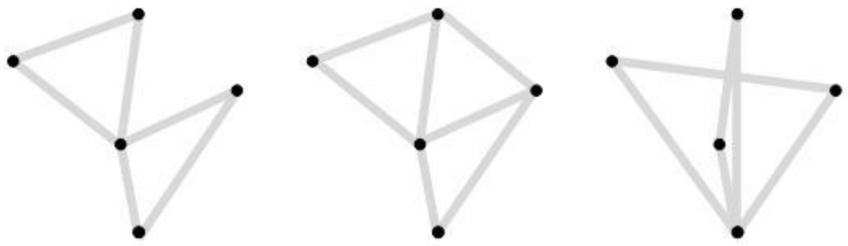


Figure 3.5: Different meshes built upon the same points

With a given polygon mesh (object of MeshGeometry3D), surfaces for a three-dimensional figure are given. Triangles are the simplest polygons that can represent a surface and are for that reason used as building blocks in WPF. Every other polygon is built upon the use of multiple triangles. In fact, the triangular mesh is the only supported mesh in WPF [7]. According to software developer and author Eric Sink, triangles are appropriate as building blocks because they meet three important requirements:

- Computational geometry algorithms become complex when they have to consider concave polygons, unlike dealing with convex polygons. All triangles are convex and therefore easier to make use of.
- All triangles are planar. With only three points you either have a plane, or some kind of degenerated triangle. There are no other cases.
- Every possible polygon can be broken up into a set of triangles.

None of these things would be true if computer graphics engines were built on any other fundamental item than the triangle [17].

3.2.2 Surfaces

Now that it is concluded that WPF uses triangles as building blocks for three-dimensional objects, there are some important concepts to consider. To be able to display a 3D object or scene, its surface has to be broken down into triangles - this should be obvious by now. The corner points of a triangle define its surface. One thing to keep in mind though is the simple fact that a 3D triangle both has a front side and a back side. The sides obviously share the exact same corner points, so there has to be some way to distinguish the front side from the back side.

The secret of defining the front side and the back side lies in the order that the triangle's corner points are given when added to the TriangleIndices property (see Section 3.2.1). The triangle side with corner points given in a counterclockwise order is defined as the front side. In order to actually show it on the screen, the *Material* property needs to be set. Likewise, to make the back side visible, the *BackMaterial* property must be assigned. If the Material or BackMaterial properties are left unassigned, the actual triangle side will be invisible. This tendency is shown in Figure 3.6; the arrow represents the sight of the viewer and the triangle's Material is set. The BackMaterial, however, is not set and therefore that side is left invisible.

This is the way to distinguish between the front and the back, and is useful if you for example want the sides to have different materials and/or colors. It is an important concept not to forget, or it may cause rendering problems in situations where the wrong side of the triangle has been set as the front side, leaving the wanted side invisible or incorrectly displayed [10] [11].

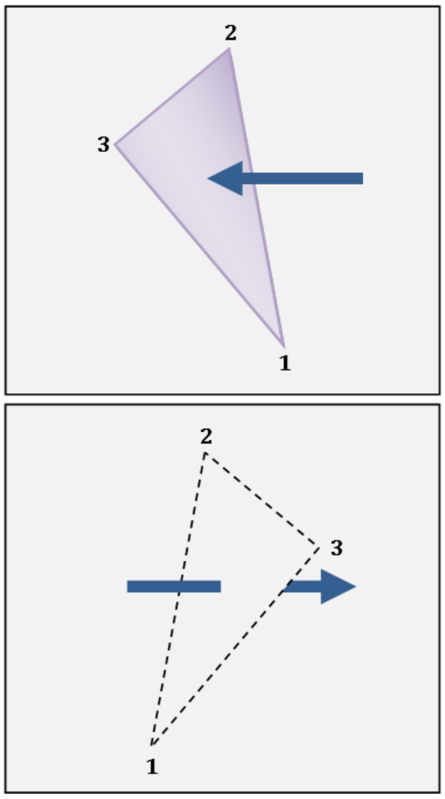


Figure 3.6: Surface visible from up front, invisible from the back

As stated in Section 3.2.1, the MeshGeometry3D object defines the shape of the figure. There is a *GeometryModel3D* class that combines the "skeleton" (the MeshGeometry3D) with the "skin" (the Material and BackMaterial). These are the three essential properties of the GeometryModel3D class and are described in Table 3.2. This is how 3D figures are defined in WPF.

Property	**Data Type**	**Description**
Geometry	*MeshGeometry3D*	Describes the shape of the GeometryModel3D.
Material	*Material*	The material used to render the front sides of the triangles specified by the Geometry.
BackMaterial	*Material*	The material used to render the back sides of the triangles specified by the Geometry.

Table 3.2: Essential properties of GeometryModel3D

3.2.3 Camera and Perspective

The 3D class library in WPF offers three different types of cameras, whereof two are of particular interest - the perspective camera and the orthographic camera. For this dissertation project the abilities of the perspective projection was preferred, because this is similar to the way the human eye works. The camera class that offers this requested ability is named *PerspectiveCamera* and has a number of properties, whereof four are of particular interest. They are described in Table 3.3 [11].

Property	**Data Type**	**Description**
Position	*Point3D*	Values for X, Y and Z are required. This property represents the location of the camera in the 3D space, but does not say anything about the look direction on which the camera's projection is centered.
LookDirection	*Vector3D*	Defines the direction in which the camera is looking.
UpDirection	*Vector3D*	The vector (0, 1, 0) is the default and means that that the top of the camera is pointing in the positive Y direction, and is not tilted in any direction considering the X and Z axes.
FieldOfView	*int*	Determines how much of the scene that can be seen at once. A low FieldOfView value will capture a smaller portion of the scene, and the displayed objects will appear large. A high value is like a wide-angle lens: a larger part of the scene is shown, but everything will appear smaller to fit in.

Table 3.3: Important PerspectiveCamera properties

3.2.4 Light Sources and Shading

The shading is calculated for each and every triangle to give a realistic impression. How the light creates shading is in part decided by *surface normals*. A surface normal is a vector that is perpendicular to a flat surface. The arrow in Figure 3.7 shows a surface normal - it is perpendicular to the surface of the triangle. Only the direction of the surface normal vector is of interest, the magnitude is not taken into consideration. The shading algorithms implemented in WPF involve these vectors. The angle between the surface normal and the direction of the light determines how the light should be reflected - that is, how the surface should be shaded (see Figure 3.8). This is calculated with the help of a shading model known as Lambert's Cosine Law, which is further discussed later in this section.

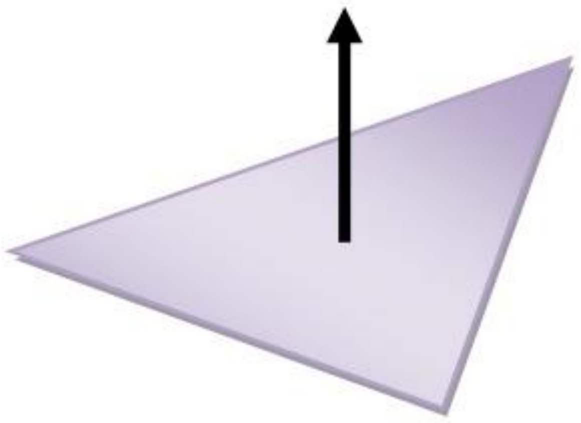


Figure 3.7: A surface normal, perpendicular to the triangle surface

The direction of light depends on what kind of light is being used. WPF offers several classes that represent different approaches to the direction of light, as seen in Table 3.4. They correspond to the four kinds of light sources previously discussed in Section 3.1.5.

Light type	**Description**
AmbientLight	The light is evenly spread in the 3D model, giving the surfaces an even shading effect.
DirectionalLight	Corresponds to distant light. The light strikes the entire area at a uniform angle defined by a three-dimensional vector.
PointLight	Corresponds to omnidirectional light. The Position property of the light source is an important aspect here, since it will determine how objects are illuminated.
SpotLight	The rays of the light are sent in different directions, but with a limited width, in the shape of a cone.

Table 3.4: Light sources in WPF

The color of the light versus the color of the surface also affects how a surface is presented. If for example the color of the light is red, then only red color will be reflected from the surface. If there is no red content in the color of the surface, then the surface will reflect no color at all and be black. This technique, setting the color of the light to determine the reflection from objects is used to set the intensity of the light. If for example the light is set to white (which is equal to RGB values 255, 255, 255), then every object reflects light with its full potential in relation to the angle of the source of the light (see Figure 3.8). For example, maximum reflectivity is achieved when the angle between the surface normal and the direction vector of the light is 180^o, because $|\cos 180°| = 1$ $(= 100\%)$. Likewise, when the same angle is 135^o, the surface reflectivity is about 71%, since $|\cos 135°| \approx 0.707$. This model is known as Lambert's Cosine Law. Furthermore, if the color of the light is set to gray (equal to RGB values 128, 128, 128), the intensity of the reflections will be half of the previous. This way, the intensity of the light is set to 50%.

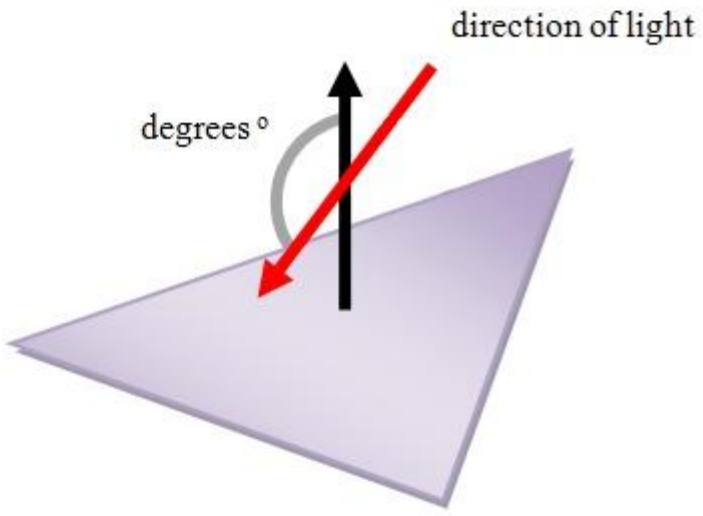


Figure 3.8: The relationship between the direction of light and surface normal

The calculation of the surface normals in relation to the angle of the incoming light is used within the local reflection model (as described in Section 3.1.5). Furthermore, a global reflection model is needed to give the impression that the scene is based on the laws of physics. This is where light-object-interaction becomes really complex. When an object's surface is hit by light energy it absorbs some of it and re-radiates some into the rest of the scene. The characteristics of the surface material determine the amount of light being reflected, together with the angle of the incoming light. Some materials absorbs almost all of the light energy (for example cotton and wool), whilst others reflect a significant amount (for example shiny metal). Each type of real world material has properties that are unique, so to be able to give a realistic impression, the real world materials has to be studied to decide their true behaviors. Making it more complex, the wavelength of the incident light also has an impact on the behavior of a given material. For materials that reflect a considerable amount of light, an objects appearance is dependent upon the position of the viewer. All of this together makes the simulation of the physical laws of light so great that the amount of processing needed is even beyond the capabilities of today’s greatest supercomputers [7].

To meet the performance goals (mentioned in Section 2.4.3) WPF makes compromises in image quality and realism. It would therefore not be an appropriate platform for creating the next blockbuster 3D animated movie, but that is not the purpose anyway. As described in Section 3.2.1, WPF lacks units of measurement. This makes it impossible to determine the size of an actual real life object that might be modeled in a scene. That is an evidence of the approximation of reality. The different wavelengths of light, as mentioned above, are not taken into consideration, nor is light attenuation or shadows. To summarize, complete global reflection models are put aside or partly approximated to achieve the performance goals. As an effect of the fact that WPF does not implement a true global reflection model, surfaces that are not directly hit by rays of light from a light source will not be lit at all and left completely dark.

3.3 Summary

This chapter has brought up some brief information about 3D graphics in general and how this concept is treated in WPF specifically. In the WPF class libraries there are lots of tools to use for setting up a 3D space, cameras, light sources, shapes, materials and other elements used in 3D graphics.

4 Product Development

This chapter will cover the implementation of the Warehouse3D product. It will introduce a first prototype that was developed during the very first weeks of the project as a means of getting to know the development environment, the concept of WPF and how to create 3D models in general.

Finally, the Warehouse3D control, which is the final product of this project work, and an application needed to demonstrate it will be presented and compared to the first prototype.

4.1 First Prototype

In order to get a feel of WPF and working with 3D models, a less comprehensive draft prototype was developed during the initial phase of the project. The goal of this was not to meet any of the given requirements, but rather to try out the WPF tool and find out what could be achieved within the scope of the project and what to leave for further development. This section will briefly cover the process of development as well as discuss the outcome of it.

4.1.1 Geometric Models

Realizing that a cylinder model would be very complex and time consuming to build and calculate solely by using triangles, an existing class library for this purpose was used. A simple rectangular surface was used as the warehouse floor and two similar rectangles served as walls. Figure 4.1 shows the first outline of the warehouse area. As discussed in Section 3.1.1, it is clearly the shading that makes it possible for a human eye to distinguish the floor from the walls. The lines show the X, Y and Z-axes of the 3D coordinate system. The area is, in this case, 50 units long, 20 units wide and 10 units high. As discussed in Section 3.2.1, this does not mean that the area is 50 pixels long (or any other unit) - it rather just determines that the model is five times longer than its height, and two and a half times longer than its width, and so on.

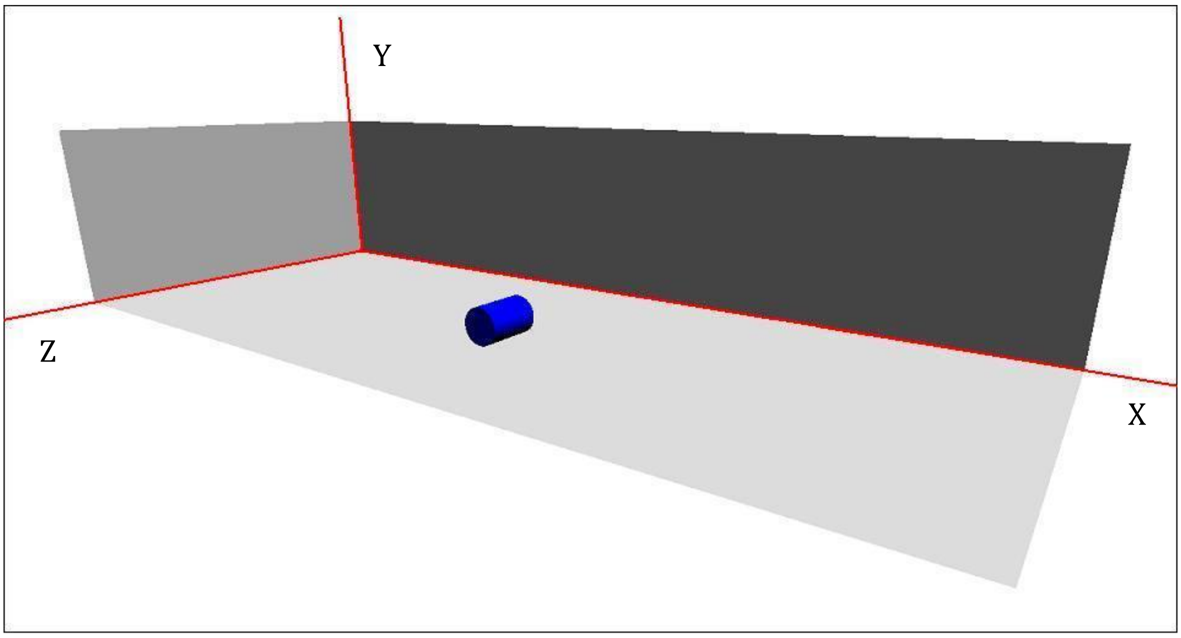


Figure 4.1: Graphical outline of the warehouse area

Just as easy as drawing the warehouse area, a cylinder model can be placed in the 3D model as well. The coordinates of the centers of the two circular short sides are given to define where it should be located. In the case of Figure 4.1, the first point (on the back side of the cylinder) is $(25, 1, 10)$ and the other is $(25, 1, 13)$. This places the cylinder in a lying position, parallel to the Z-axis. In practice, the Y-value indicates the distance from the cylinder point to the floor, so in order to draw the cylinder lying on the floor, the Y-value of its coordinates must be equal to its radius. In comparison, a coordinate pair of $(25, 0, 10)$ and $(25, 3, 10)$ would instead draw the cylinder in a standing position, parallel to the Y-axis. In this case, of course, the Y-value can be set to zero since the point on this cylinder is actually "touching" the floor.

To bring some more life and depth to the view a simple horizon was added, which could be controlled and toggled on and off with a boolean property. Its implementation was quite simple - just a huge, green 3D surface which represents the ground, and the user control background color set to blue, to serve as a sky backdrop. Also, instead of painting the floor and the walls with plain colors, a stone texture was used for this purpose which also gave additional life to the view.

There is one problem that slightly disturbs the feeling of three dimensions in Figure 4.1; there is apparently a light source in the model, but the coil on the floor does not cast any shadows in any direction, as it should do. Because of this, it is not completely obvious where exactly in the warehouse the coil is located - it can be seen either as lying on the floor in the centre of the room, or in the region of an imaginable wall where the floor ends, floating in the air. Shadows are a fundamental part in 3D graphics, but unfortunately also an extremely complex issue. This tendency is brought up in Section 5.2.3 where encountered problems are discussed.

4.1.2 Camera and Lights

A basic DirectionalLight (see Table 3.4) was used as light source. This was chosen to make sure that the corners and shapes would appear clearly, with distinct shading. At the time being, this type of light was also (incorrectly[2]) thought of as necessary in order to get as big part as possible of the warehouse enlightened without losing the shading effect. A few different directions of light were compared, but the exact value of these were not really important as long as the light fell from somewhere above, as an imaginary sun.

Some experimentation with camera positioning and movement was also included in the first prototype. Simply by rapidly assigning new positions to the camera in a repeating fashion, the user gets the impression of the camera being continuously moved along a given axis. This was the first introduction to camera movement during the project.

4.1.3 Consuming Application

For the sake of simplicity for a first outline, the warehouse overview and the consuming application was implemented as one single piece instead of two separate entities. To let a user control the camera, three sliders were implemented and their numeric values were simply assigned to the camera position’s X, Y and Z values[3]. In addition to this, some keyboard shortcuts were implemented for placing the camera in a number of predefined positions around the model.

4.1.4 Conclusions

Having developed this first prototype, some conclusions could be drawn: by getting some hands-on experience of WPF and how to display 3D shapes on the screen, realistic time estimation could be made. The limitations of the project could also be set (see Section 1.1), defining what functionality to include and what to skip. In fact, no part seemed to be so difficult that it could not be included in the project. It was decided that a demo application

[2] Later on, it turned out that multiple different light sources could be combined to get an even better illumination effect – more on this in Section 5.2.5.

[3] A slider control works with numeric values and has a min value, a max value and a current value in between. Whenever the current value is changed - that is, when the user drags the slider in either direction - an event is raised. This is a handy control for forcing the user to enter a numeric value in a quick and easy way.

should be developed at the very end, when all functionality in the Warehouse3D control should be in place and working. It was also assumed that this should be the least advanced part of the project, since it would not include any logic at all in itself. Implementing camera movement, on the other hand, were thought of as being the most difficult part to do, as it most likely would include quite a lot of advanced mathematics.

Drawing simple figures like triangles, rectangles and cubes turned out to be relatively easy. Some preparatory sketching with pen and paper were needed in order to keep track of the points and their mutual order (see Section 3.2.2). However, cylinders and other rounded shapes were much more difficult and demanded quite complex algorithms in order to calculate all points and surface normals. Because of this fact, a small existing class library, developed by Microsoft veteran Charles Petzold, was used and slightly altered to fit the purposes of the current project.

The result of the first prototype showed that the essential requirements (see Section 1.1.1) were fully affordable to achieve. The estimations suggested that the extra functionality needed in order to fulfill the additional requirements (see Section 1.1.2 and 1.1.3) could be built upon this first frame without any major problems. However, some reconstruction and redesigning of the classes were needed to keep the object graph loosely coupled and easy to extend. Therefore, the first prototype was abandoned, but served as a model for the final product.

4.2 Final product

When the first prototype (see previous section) was completed, the work with the final product started. This section will cover the development phase of this, as well as some comparisons with the first prototype.

4.2.1 Overview

The final product was implemented as a WPF User Control - an entity made up of a number of constituent controls bound together by a common functionality in a shared user interface [18]. This means that the product cannot be used “on its own” - it needs to be implemented as part of a consuming application - but in return it will be reusable and extensible in many different projects and environments. Its behavior can easily be altered with its public properties and methods.

4.2.2 The 3D World

Even though the focus should be on the warehouse and not on the surroundings, the default all-white background might feel a bit dull. Therefore, the green and blue horizon used in the first prototype (see Section 4.1.1) was implemented. This would give the user a feeling of a "real world" in the model, even though it is strictly cosmetic and has nothing whatever to do with the functionality. In order to be able to show a more realistic background, a second horizon was added. The three different backgrounds implemented can be seen in Figure 4.2.

The added horizon uses the same kind of ground surface as the other one, but instead of just using a green color, a photo was used as a texture. As for the sky, things got a little more complicated. Just like the ground, a photo was meant to be used as a texture. However, that texture would need a surface, and not just a simple "wall" in the background, since that would give the impression of standing in an enormous room rather than being outdoors. A huge hemispheric shape enclosing the warehouse model might seem like a natural choice, but was skipped because of two reasons; first, it would be a quite complex shape to render for this purpose, considering this would be just a nice feature and not something to spend lots and lots of time on developing. Second, a rectangular image texture stretched over such kind of a shape would have been distorted and thus not very realistic. Therefore, a low cylinder with a huge circumference was used, forming a circular wall around the warehouse model. The concept of this is depicted in Figure 4.3, with a warehouse model in the center (not to scale).

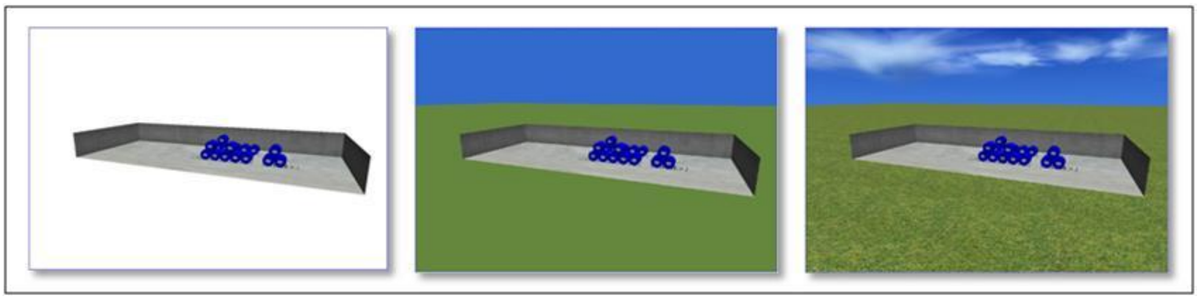


Figure 4.2: The different backgrounds used in the final product

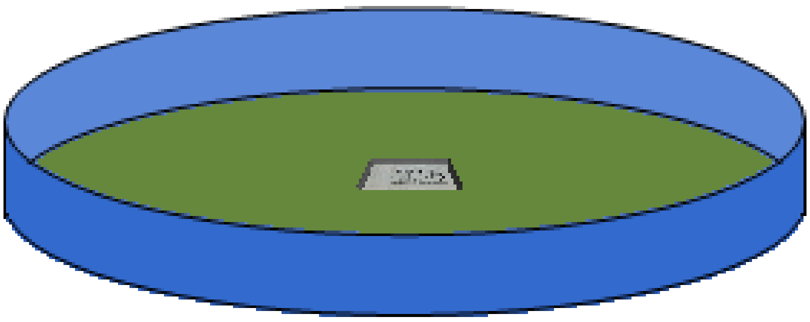


Figure 4.3: Concept of the cylindrical horizon

4.2.3 The Warehouse

The main idea for the warehouse area was taken directly from the first prototype, with some changes and improvements. For versatility, the size and shape of the warehouse were exposed as public properties. Controls were also set to swap the length and width property values in case the width value entered would be greater than the length value. This ensures that the warehouse is always drawn in the same direction in the graphical view. This goes only for rectangular shapes of course - it was soon discovered that the idea of non-rectangular shapes (see the requirements in Section 1.1.3) would be very complex and therefore the implementation of this was postponed and, ultimately, completely abandoned. Further discussion on this problem can be found in Section 5.2.2.

In the first prototype, the idea was to always let the user see inside the warehouse without the nearest walls blocking the line of sight, but still having the farthest walls visible as some sort of "backdrop". Because of this, only two walls were drawn (see Section 4.1.1). However, this brings up the question: what if the camera is placed on the opposite side of the warehouse and turned 180 degrees, what will happen then? No walls would be seen at all - the two drawn walls would become invisible because of the behavior of 3D surface drawing in WPF, as mentioned in Section 3.2.2. One possible solution to this would be to always keep track of the camera position to make sure the "correct" (i.e. farthest) walls are displayed and the nearest ones are hidden. This would have its obvious drawbacks though; lots of code and overhead. The solution of choice was to actually draw all walls, but only the back sides, taking full advantage of the fact that sides of a surface left unspecified will be invisible.

4.2.4 Warehouse Items

One of the project requirements was to be able to use other geometric shapes (see Section 1.1.2) than cylinders. In addition to cylinders, box shapes has also been implemented and can be used as warehouse items. Except for these two, no other classes have been created, but can be added and used as long as it implements the interfaces needed, as presented in Section 5.1.1.

Each item positioned on the floor in the warehouse should be able to have its own slot, according to the project requirements. The slots are displayed as black squares and can contain a string value with an address, ID or similar. A user can of course set the position and size of each item slot. It is important to know that the slots merely have a cosmetic purpose – a single slot is only a graphical feature and cannot hold any reference to its associated item.

Likewise, labels can be added to a coil. Just like the mentioned slots, these are also displayed as squares and can show a name or ID of the coil. This was a “nice to have”-feature in terms of requirements and has no impact on the overall functionality whatsoever.

4.2.5 Camera Positioning

Because of the behavior of the light source in the 3D model (which was defined as a DirectionalLight, see Section 3.2.4) parts of the warehouse and its coils are left dark, due to the fact that the light cannot reach these areas. Therefore, there was no real reason in showing the warehouse from those angles, since it is difficult to discern the coils from each other and it would not give the user any useful information. Seven fixed camera positions were defined, as described in Table 4.1.

The distance between the camera and the warehouse was calculated to be as short as possible, but long enough to fit the entire warehouse on the screen. This was easily done with the help of some trigonometric calculations. By using the given FieldOfView property of the camera along with the already known length of the warehouse, an isosceles triangle could be defined, as shown in Figure 4.4, where the vertex angle α is equal to the FieldOfView property of the camera.

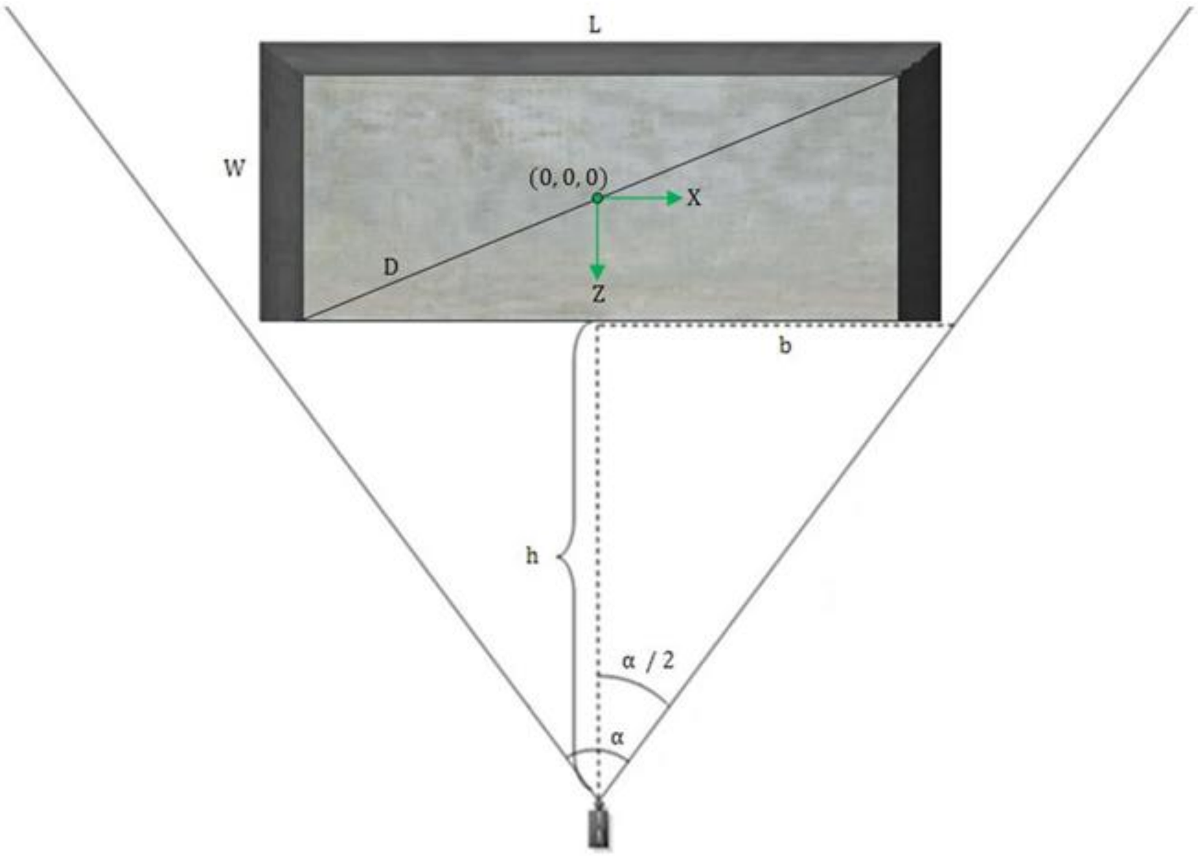


Figure 4.4: Calculations of camera position

The triangle side b is defined as

$$b = \frac{1.05L}{2}$$

where the constant 1.05 is used to include some extra space and L is the length of the warehouse. Knowing these values, the distance between the camera and the warehouse, h, can easily be calculated as

$$h = \frac{b}{\tan\frac{\alpha}{2}}$$

Given that B (not shown in Figure 4.4) is the base altitude of the camera, representing the height of an imaginary camera tripod, this gives us the fixed camera position $(0, \mathrm{B}, \mathrm{h} + \frac{\mathrm{W}}{2})$, denoted "South" in Table 4.1. Similar calculations are made for the east and the two top camera positions.

The values used in the calculation of the PerspectiveSouth position are the same values as those used for the South position, except that they are used in different ways. For the PerspectiveSouth view, the Z-wise distance between the camera and the warehouse is halved, and instead the altitude of the camera is raised to the value of h. This means the PerspectiveSouth camera position is set to $(0, \mathrm{h}, \frac{\mathrm{h}+\frac{\mathrm{W}}{2}}{2})$, since it turned out that this position served its purpose very well for most warehouses, no matter their size. As for the PerspectiveSouthEast and PerspectiveSouthWest positions, a different approach is taken:

Consider $\vec{D}$ to be a vector representing the diagonal of the warehouse area, running from the back right corner to the front left corner (see Figure 4.4). $\vec{D} = (-L, 0, W)$. The normal $\vec{N}$ to this vector can then be defined as $\vec{N} = (W, 0, L)$. The LookDirection property of the camera is set to $-\vec{N}$ for the PerspectiveSouthEast position in the XZ-plane, and the Position property of the camera (which is of type Point3D type, see Table 3.3) is set to N. This makes sure that the camera is pointing straight at the center of the warehouse.

At this point, the camera position in the XZ-plane is determined. Next, concerning the Y value, it should be set to a value depending on the length of the warehouse under the idea that the longer the warehouse, the greater the altitude of the camera in order to get a good overview of the entire warehouse. Therefore, P_y is simply set to L.

The calculation should be completed at this point. P has been set to (W, L, L) and should work out fine as the camera position. However, tests showed that these values were a bit too large, i.e. the camera was placed a little too far away from the warehouse. To solve this, some values were tried out as a multiplier, and 0.6 seemed to be the magic number of use, regardless of the warehouse size[4]. Therefore, the Position property of the camera in the position denoted “PerspectiveSouthEast” (see Table 4.1) was finally set to:

$$\mathrm{P} = (0.6\mathrm{W}, 0.6\mathrm{L}, 0.6\mathrm{L})$$

Of course, the position denoted “PerspectiveSouthWest” is simply a mirrored version of the “PerspectiveSouthEast” position.

[4] As long as the warehouse has a reasonable size - it is always impossible to get a good overview of an enormous warehouse.

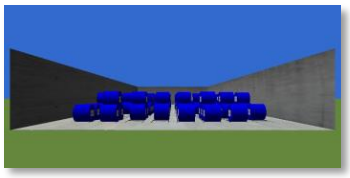
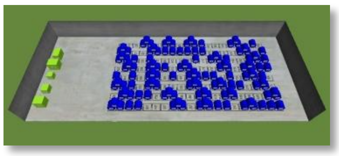
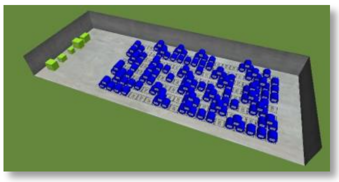
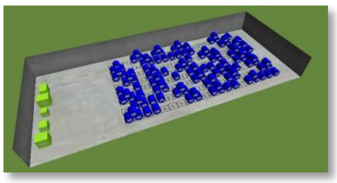
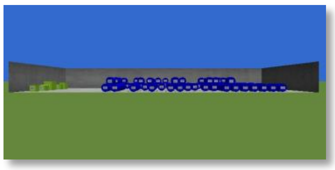
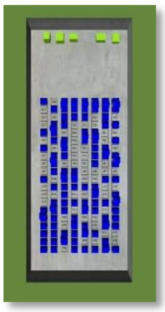
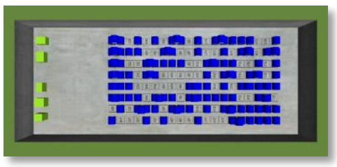
Camera Position	Example	Description
East		The camera is placed on the eastern side of the warehouse, pointing towards its short side.
PerspectiveSouth		The camera is placed on the south side of the warehouse, slightly elevated.
PerspectiveSouthEast		The camera is placed on the southeastern side of the warehouse, slightly elevated.
PerspectiveSouthWest		The camera is placed on the southwestern side of the warehouse, slightly elevated.
South		The camera is placed on the south side of the warehouse, pointing towards its long side.
TopLength		The camera is placed above the warehouse pointing straight down. This is a true birds-eye view and as near a 2D view as possible.
TopWidth		As TopLength, but with the camera rotated 90° clockwise.

Table 4.1: Fixed camera positions

4.2.6 Camera Movement

According to the project requirements in Section 1.1.2, the user should be able to move the camera freely in all directions to see the warehouse view from almost any angle (perhaps except from below). This turned out to be by far the most complex part of the whole project and is discussed in more detail in Section 5.2.1.

The more control a user has over the camera, the more complex it becomes to maneuver it. For example, there is a risk that the user accidentally turns the camera upside down and must waste time trying to get it right again. To make sure this would not happen, the camera movement was strictly limited. Starting from one of the positions mentioned in Section 4.2.5, the camera can move straight along its own relative X, Y or Z axes. Here, the axes are not the axes of the world coordinate system, but instead a relative coordinate system fixed to the camera. For example, when the camera is in one of the fixed top positions (see Table 4.1), its relative Z-axis is actually equal to the Y-axis of the world coordinate system. Some limits were set to keep the camera within reasonable bounds in relation to the warehouse.

Public methods for moving the camera one thousand units back or forth along its X, Y or Z axis were implemented. According to Sogeti, units would probably represent millimeters in most environments of use, which is the reason for using thousand-unit steps in camera movement. This worked out fine, but in retrospect it would probably have been a better idea to move the camera in steps relative to the size of the warehouse dimensions, in a similar way as the camera positions were calculated (see Section 4.2.5).

4.2.7 Demo Application

As described in Section 2.3.4, a small stand-alone application implementing the Warehouse3D control was needed in order to demonstrate the product. Its only purpose is to show the functionality of the product and nothing else. Therefore it is kept quite simple, with a minimum of code.

When the application starts, the user can add a warehouse to an empty “world” and add a predefined set of empty item slots. According to the original idea as described in Section 2.3.4, the user should be able to manually add an item to the warehouse. However, this functionality was modified because of two reasons - first, it would take a considerable amount of time to manually add the items one by one. Second, following this approach would not reflect the original idea of reading the data from a separate data source and fill the warehouse.

To simulate the data reading, the user can instead generate the content of the warehouse with one single button click. When doing this, a number of coils are placed in the slots, and a number of boxes are placed along one of the short walls. Virtually all values within this procedure are randomized; the number of items, their position, the size of the boxes and also all individual information about each item (see below). Figure 4.5 shows an example of what such a randomized warehouse might look like.

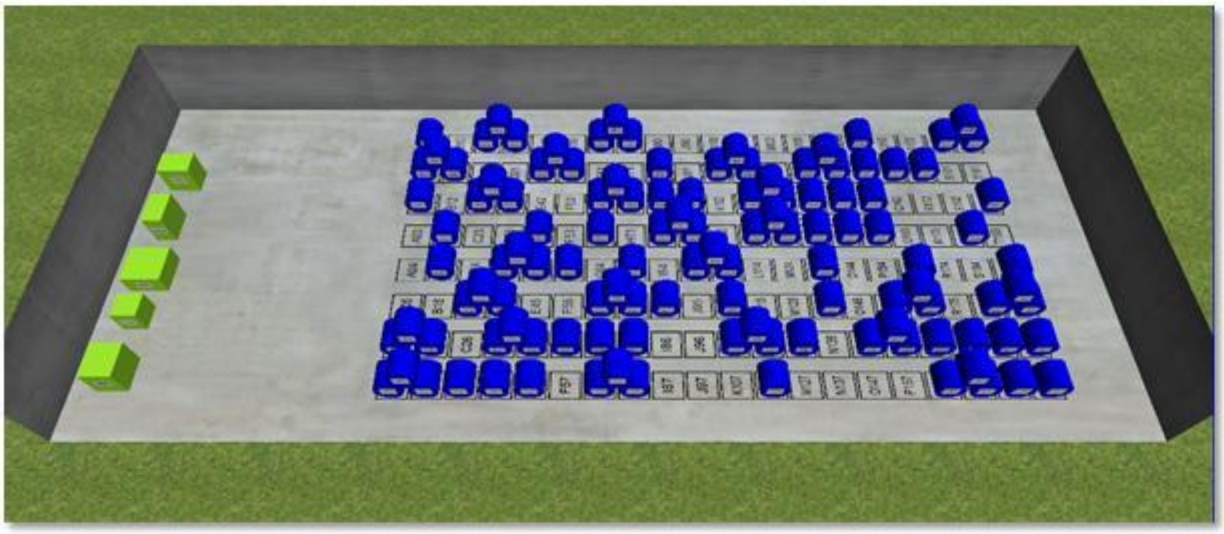
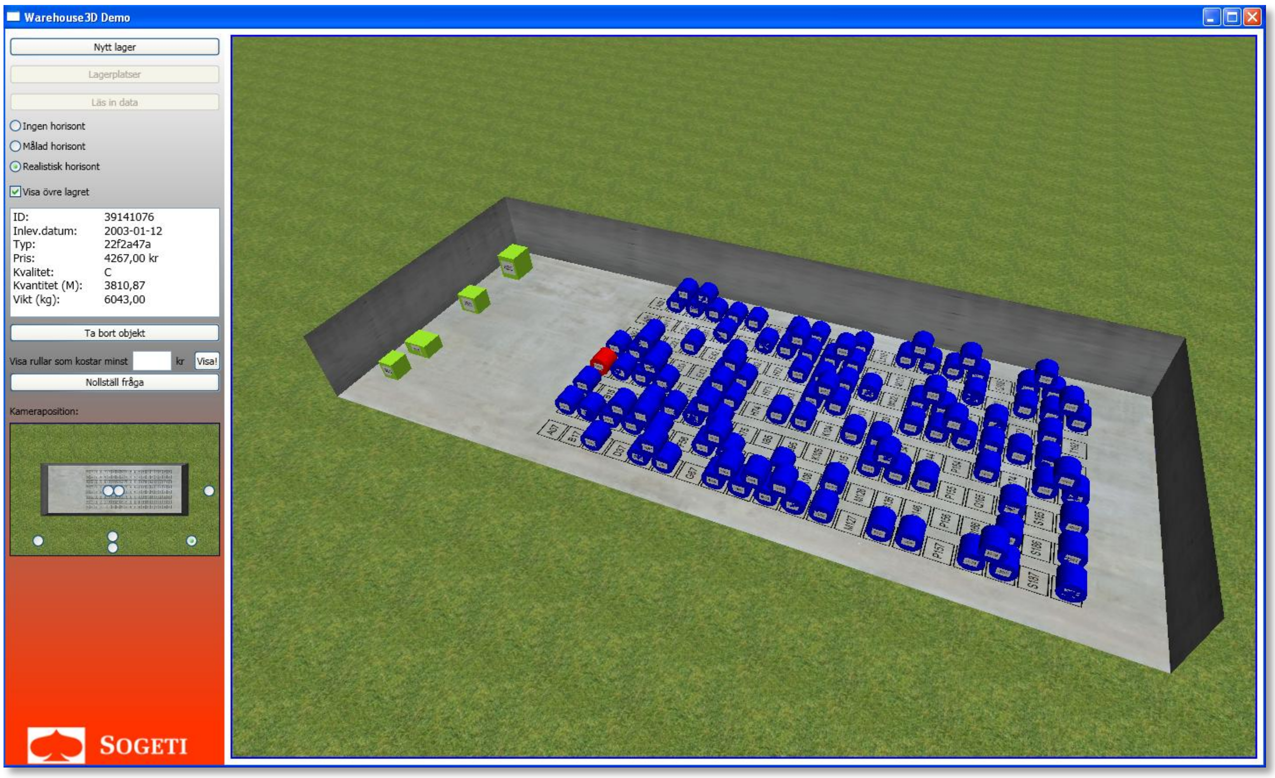


Figure 4.5: Warehouse with randomized content

By clicking on one of the items, some individual information about the specific item is shown in a text box in the demo application. These values are for example ID, date of delivery, price and weight. In this particular case they are just nonsense values with the only purpose to demonstrate what can be achieved with the product. If the CTRL key is held down, the user can select several items, just like when selecting files in Windows. When more than one item is selected, the individual values are added and the sum is shown in the information text box.
There are also three radio buttons which can be used for switching between the three different horizon types (see Section 4.2.2). The user can also choose to hide the second level of items in order to see the underlying ones more clearly. There is also a panel for changing camera position and a button for removing an item from the warehouse permanently. Finally, the keyboard is used to move the camera around in the 3D space. Figure 4.6 shows a screenshot of the demo application and the Warehouse3D-control.



( ) Ingen horisont
( ) Målad horisont
[✓ Realistisk horisont]
[✓ Visa övre lagret]

ID:	39141076
Inlev.datum:	2003-01-12
Typ:	22f2a47a
Pris:	4267,00 kr
Kvalitet:	C
Kvantitet (M):	3810,87
Vikt (kg):	6043,00

Visa rullar som kostar minst ____ kr

Figure 4.6: Demo application

4.3 Summary

The implementation of the Warehouse3D product has been discussed. This includes both a first prototype which was built to get some experience with the development environment, and the main project which eventually resulted in the final product. An application to test and demonstrate this final product was also developed as a last part of the project.

5 Results

This chapter will present the results of the final product, both in itself and compared to the expectations from an early stage and the given requirements. The problems encountered during the development phase will also be brought up and explained.

5.1 Technical Outcome

In retrospect, most of the requirements (see Section 1.1) were met with the final product. The warehouse can be viewed from straight above and several three-dimensional views. Adding and removing coils can also be done, and a demo application has been developed to demonstrate this. Overall, this means that the essential requirements (see Section 1.1.1) are fully met. Note that there is no built-in validation of where coils (or other items) are placed, which means it is possible to put items outside the warehouse or even floating in the air. However, since Warehouse3D just serves as a view for some underlying data, it is ultimately up to this data source to keep its values valid.

The architecture of Warehouse3D has been developed with the aim to keep the classes loosely coupled. This makes the product open for extensions and not limited to cylinder shapes only. It is hard to predict its future usage, but hopefully it is sufficiently scalable to be used in several different projects. The class diagram seen in Table 5.1 shows the relationship between the classes used in Warehouse3D, and can be a useful resource for future developers who want to add more functionality to the product.

Similar to selecting files by clicking them in Microsoft Windows, a user can select one or more items in the warehouse in just the same way. There are also public methods for hiding and showing items in the warehouse. Also, item slots of any size can be added anywhere in the warehouse and, just as with warehouse items, they can be placed outside the warehouse. However, these slots are stuck to the ground so, unlike cylinders, they cannot be placed floating in the air. All in all, this means that the important requirements (described in Section 1.1.2) are fully met, except for the last point: camera movement. This part turned out to be a bit of a problem and does not completely match the original requirement. Section 5.2.1 discusses this issue further.

Finally, "sticky" documentation labels can be fixed onto a warehouse item, showing a name or an ID of the specific item. This feature was one of the two points under "Further development" (see Section 1.1.3). However, the other point - to be able to display non-rectangular warehouses - was skipped because of the troubles and difficulties it entailed (more on this in Section 5.2.2).

5.1.1 Architecture

A number of classes, interfaces and enumerations have been created in the project (see Figure 5.1). Representing the graphical view, the most important class is arguably *Warehouse3D,* which inherits directly from the built-in .NET class UserControl. This class exposes a Warehouse-class (through the IWarehouse[5]-interface) which is responsible for keeping track of all warehouse data such as its size, items contained - both geometric shape items such as cylinders or boxes (classes implementing IWarehouseItem) and the item slots on the floor (the ItemSlot class) - and a collection of currently selected (highlighted) items, which can be seen as a subset of the items contained. In code, this is represented by two IWarehouseItem lists, Items and SelectedItems, where the latter is initially an empty list. Whenever a warehouse item is clicked (and thus selected), the appropriate object is copied from the Items list to the SelectedItems list. Similarly, when an object is deselected, it is removed from the SelectedItems list.

Since both the item slots and the item labels (see Section 4.2.4) in essence are simple squares with text inside, they both inherit from the abstract *TextBox3D* class. .

The abstract *BaseInfoObject* class is used to store information about a specific item, for example its price, weight, material or whatever information that might be useful. The BaseInfoObject class itself only has two properties; ID represents a name or identity of the item, and Layer stores a value that indicates which Y-wise layer in the warehouse the item can be found in. Besides these two properties, which apply to all items in the warehouse, the other information may vary between different implementations and companies using this product. Therefore the developers have to define their own info classes for this purpose which must inherit from the BaseInfoObject class. In this particular project, a simple info object class (called TestInfoObject) was created and used in the demo application for storing price, weight and information about date of delivery (as mentioned in Section 4.2.7).

The *WarehouseCylinder* and *WarehouseBox* classes included in this project can be used immediately. However, just like the case with the info classes, if another shape is desired, the developer has to create a class for this that implements the IWarehouseItem

[5] Microsoft's naming standard suggests that interface names should begin with a capital I.

interface, and inheriting some shape class which, in turn, inherits from the abstract ModelVisualBase class.

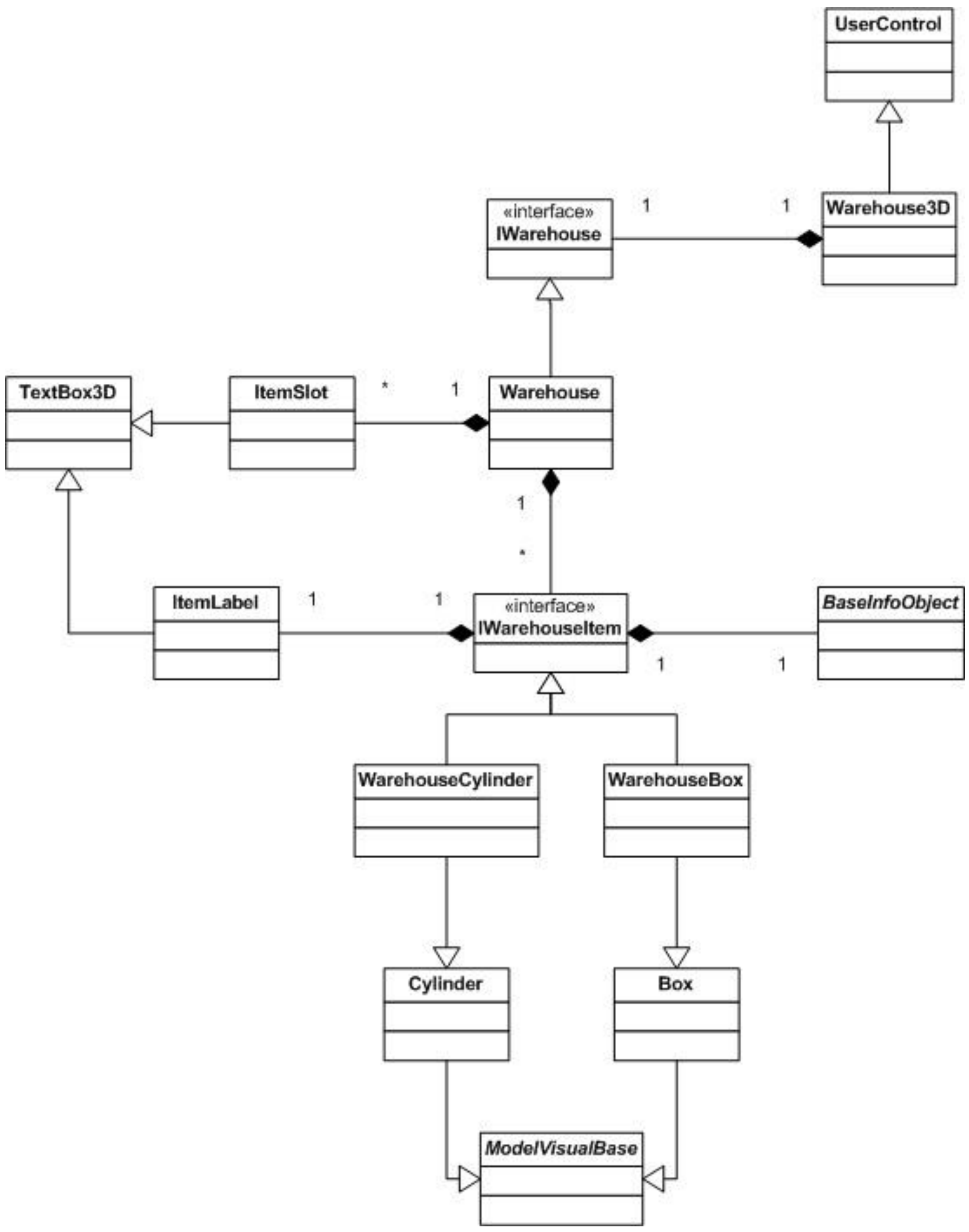


Figure 5.1: Class diagram

5.1.2 Using the Warehouse3D Control

To use the final product, a consuming application must implement it. This application should preferably be created in WPF, but with a few preparations and settings, a regular Windows Forms application could also work. Given there is a container control (a Form, a Panel or similar) in the consuming application, inserting a Warehouse3D control into the consuming application can be done with just two lines of code:

```
Warehouse3D myWarehouse = new Warehouse3D();
container.Children.Add(myWarehouse);
```

To make further adjustments to the Warehouse3D control, such as entering a size of the warehouse, add items, change the horizon, etc, methods on the myWarehouse-object would be used. For more information about this, see Appendix A.

5.2 Problems

During the development phase, some problems were encountered. This section will cover these issues and the approaches through which they were solved, if solved.

5.2.1 Camera Movement

One of the important requirements (see Section 1.1.2) was to allow three-dimensional camera movement. Initially the goal was to allow the user to freely “float around” the model and to be able to view the warehouse from virtually any possible angle and distance. When searching for information on how to implement such movement of the camera, an existing solution for a “virtual trackball” [19] was found. It gives the user the opportunity to rotate the model as if it was placed inside an invisible sphere. However, to this particular model, the trackball solution was not completely what was sought after. It would sure give the user the opportunity to freely “float around”, but it would also be a little too easy to accidentally turn the model upside down. Once in such position, it was not completely intuitive to get the model “back on track” in an upright position again. Since the Warehouse3D control very well could be used by people with nearly no computer experience at all, some limitations to the movement in order was necessary to prevent the user from putting the model in an unwanted position within the sphere. Therefore, different solutions were tried out to alter the existing “trackball”, setting limitations in the allowed movement. This task turned out to be trickier than expected.

As a result from a discussion with Thomas Heder, the project supervisor at Sogeti, it was agreed that predefined starting positions for the camera should be used, from where the user would be able to pan the camera sideways and up and down, as well as zooming in and out. This way the user cannot turn the camera and may not get in a position where the model is upside down or similar. The chosen predefined camera positions are summarized in Table 4.1.

5.2.2 Non-rectangular Warehouse Areas

According to the further development requirements (Section 1.1.3), the intention was to be able to create warehouses of non-rectangular shapes. This would include all kinds of possible warehouse shapes, for example warehouses shaped as the letter “L”, “T”, or even more advanced ones like an “E”. In the early part of the project, efforts were made to produce an algorithm that would create a desired warehouse area, with the positions of all the corners as input. It did not behave appropriate though, and in the end of the project, along with the fact that this part belonged in the category of extra “nice to have”-features, this functionality was abandoned. Nevertheless, it is still a useful feature, and experiences and thoughts on this are presented further in Section 6.3.2.

5.2.3 Shadows

Shadows are an essential part of a three-dimensional environment, but a complex part as well. Real light based shadows are not implemented in WPF [20]. This fact was realized early on, and alternate ways to get the objects to cast shadows were looked into but unfortunately, no good solutions for this were found. However, the fact that shadows do not exist in the model does not have a major impact on how the scene is perceived, even though it would have been clearer with the use of shadows. An example of this problem can be seen in Figure 4.1.

5.2.4 Window Resizing

When experimenting with different sizes for the viewport window, the content in the window behaved differently depending on whether the window got wider or narrower, or whether it got taller or shorter. When making the window narrower or wider the whole content changed proportionally in size, whereas making it taller or shorter did not change the content size. What happened was that the content was clipped if its height was not enough to display it. In other words, the size of the 3D figures depended in part on the width of the viewport window.

The reason why the behavior was different when the height and the width of the window were altered depended on the FieldOfView property of the camera. This property determines how much of the scene that is to be shown, independent of the width of the viewport window. To fulfill this, the sizes of the objects as we see them, are altered to be able to display them all, independent of the window width. Concerning the height of the window, there is no property to determine what is to be seen, so the content is clipped instead of altered in size [11].

As the reasons for the behavior were realized, there was not much of a problem. An almost quadratic window size was used to make sure that the scene would look good under these circumstances.

5.2.5 Lighting with Multiple Light Sources

Initially, only one light source (DirectionalLight) was used. However, in order to both get shading effects on the objects, as well as an overall light with a weaker light intensity to get at least some illumination even at the darkest areas, different light sources were combined. This led to a lighting model where some non-parallel surfaces got the same amount of shading, which made the affected surfaces look somewhat smudgy, as if they were colored with oil crayons.

This was an effect of the fact that a particular surface only can get illuminated to a particular level. That level is reached when the surface is reflecting at its full capacity. With the two light sources, the surfaces fully facing the directional light got maximum illumination from the directional light, but also got additional illumination from the ambient light as well. The effect of the ambient light did not make any difference, as the surface was already fully illuminated. The adjacent surfaces were affected though. They may have got 90% lit by the directional light, giving them a certain shading effect, but when the ambient light added up, they were enlightened over 100%. As 100% is the maximum, these surfaces got lit in the same manner as the surfaces completely facing the directional light. The effect of this was that non-parallel surfaces got lit the same way. The curved cylinders got affected by this and appeared smudgy.

The solution was to make sure that only surfaces at a given angle could get full illumination - in this case it would be the surfaces completely facing the directional light. Therefore, the directional light was assigned an intensity of 75% and the ambient light an intensity of 25% (light intensity was discussed further in Section 3.2.4), for a combined maximum illumination of 100%.

5.2.6 Horizon Adapting to Large Warehouses

The concept of the realistic horizon (implemented as a large cylinder) was discussed in Section 4.2.2. It adapts to the size of the warehouse modeled in the scene. There is not just the size of the cylinder to adjust though. A sky image was used as a texture, stretched to fit the cylinder properly. If the cylinder gets very high and/or wide, the image could get stretched in a manner that is not appealing. For warehouses with non-extreme shapes it does not appear as a problem, but as the warehouse gets very wide or high, the approach of the horizon becomes less realistic.

5.3 Evaluation

The feedback was overall positive when presenting the project result to the employees at Sogeti. The product demonstration was followed with enthusiasm and triggered curiosity and questions on usage and possible further development. Also, the graphical interface seemed to draw people’s attention in a positive way.

5.4 Summary

The results of the project have been discussed throughout this chapter. The final product has been evaluated and compared to the given requirements (see Section 1.1). One of the lowest prioritized requirements - to handle non-rectangular warehouse shapes - was skipped, and the camera movement feature was slightly altered. Apart from these points, all initial requirements were met.

Problems encountered during the development phase have also been brought up in this chapter, along with the solutions to the problems that were solved. The majority of the problems were not directly connected to the project requirements, but still crucial for the best appearance possible.

6 Conclusions

The goal with this project has been to make warehouses easier to overview, with the help from three-dimensional computer graphics. Lots of useful knowledge and experience has been achieved during the project and, with that knowledge, the outcome is a result that achieves the goal.

This chapter will present a summary of the dissertation work and the experiences acquired along with ideas for future development.

6.1 The Product

A product that serves as a three-dimensional visualizing tool for a warehouse and its stored contents has been developed. It is built upon the idea that an actual database holding the locations of the stored items should serve as the input, and that the product gives a graphical overview of the stored items. A test application was also built in order to demonstrate the implemented functionality and features.

The product developed was given the name Warehouse3D, which goes along well with the already existing names that occur among the 3D types and classes in WPF.

6.2 Experiences

Experience from working in WPF (more described in Section 2.4.3) was acquired during the course of this project, as well as some knowledge on how 3D computer graphics are treated in general, and in WPF particularly. Having some experience from Microsoft developing and C#-programming, WPF was quite easy to get started with, since that knowledge could be brought into WPF as it consists of both “traditional” CLR compliant code (such as C#, see Section 2.4.1) and XAML markup.

This dissertation project has been exciting to work with, and the outcome of the final product is satisfactory for both Sogeti and the authors of this thesis.

6.3 Further Development

Different ideas for further development have come up during the course of the project. These are features that were not covered by the requirements for this dissertation (except for one -

see Section 6.3.2), but are still interesting and would probably be very helpful in an extension. This section covers these ideas.

6.3.1 Warehouse Item Positioning

Our warehouse items can be placed anywhere in the scene. Coordinates are given for center of the back side of the object, and the item is placed accordingly. But there is a limitation - the item itself is placed lengthwise along the fixed Z-axis of the world coordinate system in the scene. There is always a possibility that items are placed in a different position, but the product will display the items in the same manner. To be as accurate as possible, the product would do well if it was able to determine how the individual warehouse items were positioned.

6.3.2 Different Warehouse Shapes

This is the one functionality that was initially covered by the requirements for the product. It was part of the Further Development requirements (see Section 1.1.3). Since this requirement was not fulfilled, it is put as a suggestion for further development. It is highly recommended to develop this functionality, since warehouses might not always be rectangular or quadratic. The development will eventually face some issues though: besides the complex algorithm needed to calculate the shape of the warehouse, another problem may occur when creating an area in the shape of the letter “E” for example. In the rectangular warehouse, advantages were taken of the fact that non-specified sides of triangles appear invisible (as discussed in Section 3.2.2). But with an “E” shaped warehouse, some visible walls would be blocking the sight into other parts of the warehouse area. This might limit the warehouse overview from positions that are not located straight above. The approach to this tendency is something to consider.

6.3.3 Multiple Camera Views

When deciding the fixed camera positions that should be available, it turned out that different camera views had different advantages. An idea that came up in connection with this was to be able to show multiple camera views at the same time, to simultaneously get the benefits from different views. However, no effort was put into this, since it was not stated as a requirement for the product, but it could probably be a great functionality in the scope of a further development.

6.3.4 Shadows

Even though the Warehouse3D product is useful and does not suffer that much from the lack of shadows, it is inevitably necessary to implement shadows in order to get a realistic experience when navigation in a three-dimensional scene. Shadow algorithms are not implemented in WPF, but some other option to get shadows in the scene would preferably be considered in an extension.

6.3.5 Overhead Crane as Storage

At this stage, the warehouse items in the scene are lying visible according to the stated coordinates. There is one spot though, where an item also could be positioned - in the overhead crane (or corresponding mechanism). It would likely be of interest to keep track of the item currently positioned in the crane grip as well. To figure out how this should be considered and implemented is a possible task to look into.

6.3.6 Hiding and Showing Items

The initial idea for hiding and showing items in the warehouse was to be able to switch an entire layer between visible and invisible. This would be a useful feature, especially from the camera positions set above the warehouse (TopWidth and TopLength, see Section 4.2.5). This was also implemented with success.

However, this feature only works on horizontal layers (i.e. in the XZ-plane). It might also be interesting to be able to hide entire rows or columns (that is, a layer in the XY-plane or the YZ-plane), depending on the camera position.

6.4 Summary

In accordance with the initially set requirements, a product has been developed with a satisfactory outcome. New insights into three-dimensional computer graphics were achieved, along with knowledge and experience from WPF as a development tool. Ideas for further development that were encountered during the project were put together and presented in this final chapter.

Sogeti, serving as product owner in this project, were satisfied with the result. Judging from the demonstration, it seems like it also has a somewhat appealing graphical layout that may be used in marketing purposes. In fact, an early version of the product has already been demonstrated in a customer meeting with a positive response.

Overall, as it serves its purpose both as a technical product and as a tool for demonstration, the project work can be considered successful.

References

[1] Sogeti. *About Sogeti*. Sogeti Sverige AB. 2010. http://www.sogeti.se/sv/In-english/About-Sogeti/, 2010-12-08.

[2] Tim Patrick. *Start-to-Finish Visual Basic 2005*. Boston, Addison-Wesley 2006.

[3] Wikipedia. *.NET Framework*. http://en.wikipedia.org/wiki/.NET_Framework, 2010-10-12.

[4] Microsoft. *Overview of the .NET Framework*. Microsoft Developers Network. 2010. http://msdn.microsoft.com/en-us/library/a4t23ktk.aspx, 2010-10-12.

[5] David Chappell. *Introducing Windows Presentation Foundation*. Microsoft Developers Network. September 2006. http://msdn.microsoft.com/en-us/library/aa663364.aspx, 2010-10-12.

[6] Charles Petzold. *Applications = Code + Markup*. Redmond, Microsoft Press 2006.

[7] Andries van Dam. *Introduction to 3D Graphics using WPF*. August 2010. http://www.sklardevelopment.com/graftext/ChapWPF3D/WPF3D_CORE__Fall2010__rev-M-8.pdf, 2010-11-14.

[8] Wikipedia. *Graphics Device Interface.* http://en.wikipedia.org/wiki/Graphics_Device_Interface#GDI.2B, 2010-11-01.

[9] Microsoft. *GDI+ (Windows).* Microsoft Developers Network. 2010. http://msdn.microsoft.com/en-us/library/ms533798%28v=VS.85%29.aspx, 2010-11-01.

[10] Mark Giambruno. *3D Graphics & Animation*. Indianapolis, New Riders 2002.

[11] Charles Petzold. *3D Programming for Windows*. Redmond, Microsoft Press 2008.

[12] Alan Watt. *3D Computer Graphics*. Boston, Addison-Wesley 2000.

[13] Wikipedia. *Polygon Mesh.* http://en.wikipedia.org/wiki/Polygon_mesh, 2011-01-11.

[14] Sean McHugh. *Understanding Camera Lenses*. http://www.cambridgeincolour.com/tutorials/camera-lenses.htm, 2011-01-10.

[15] Wikipedia. *Shading*. http://en.wikipedia.org/wiki/Shading, 2010-10-11.

[16] Matthew MacDonald. *Pro WPF in C# 2010*. New York, Apress 2010.

[17] Eric Sink. *The Twelve Days of WPF 3D*. July 2007. http://www.ericsink.com/wpf3d/, 2010-10-11.

[18] Matthew A. Stoecker. *Exam 70-502: Microsoft .NET Framework 3.5 - Windows Presentation Foundation*. Redmond, Microsoft Press 2008.

[19] Daniel Lehenbauer. *Rotating the Camera with the Mouse*. December 2005. http://blogs.msdn.com/b/danlehen/archive/2005/12/15/rotating-the-camera-with-the-mouse.aspx, 2010-12-07.

[20] .NET Development Forums. *WPF 3D - shadows*. September 2009. http://social.msdn.microsoft.com/Forums/en/wpf/thread/7fb0a541-dcd7-41f2-8e01-afa0235b3704, 2010-12-08.

A Warehouse3D Manual

Warehouse3D Class

Represents a Windows control to graphically display items in a warehouse.

Syntax

```
public class Warehouse3D : UserControl
```

Constructors

Name	Description
Warehouse3D()	Initializes a new instance of the Warehouse3D class with no arguments.
Warehouse3D(int, int)	Initializes a new instance of the Warehouse3D class with the specified length and width.
Warehouse3D(int, int, int)	Initializes a new instance of the Warehouse3D class with the specified length, width and height.
Warehouse3D(IWarehouse)	Initializes a new instance of the Warehouse3D class using the specified IWarehouse-object.

Properties

Name	Description
Camera	Gets the Camera used in the 3D model.
CameraBaseAltitude	Gets or sets the base altitude of the camera, i.e. the height of an imaginary camera tripod.
CameraLocked	Gets or sets a value indicating whether the camera can be moved or not.
CameraPosition	Gets or sets the camera position.
Horizon	Gets or sets the type of horizon used in the 3D model.
Warehouse	Gets or sets the warehouse used in the 3D model.

Methods

Name	Description
MoveBackwards	Moves the camera 1000 units backwards.
MoveDown	Moves the camera 1000 units downwards.
MoveForward	Moves the camera 1000 units forwards.
MoveLeft	Moves the camera 1000 units to the left.
MoveRight	Moves the camera 1000 units to the right.
MoveUp	Moves the camera 1000 units upwards.
ShowAllLayers	Shows all item layers in the warehouse.
ShowLayer(int, bool)	Shows or hides the specified item layer.

Events

Name	Description
ItemClicked	Occurs when an item in the warehouse is clicked.

IWarehouse Interface

Defines properties for Warehouse classes to use in the Warehouse3D control.

Syntax

```
public interface IWarehouse
```

Properties

Name	Description
BackLeftCorner	Gets the point of the back left (northwest) corner of the warehouse.
BackRightCorner	Gets the point of the back left (northeast) corner of the warehouse.
Diagonal	Gets the diagonal of the warehouse.
DownLimit	Gets or sets the minimum Y value allowed for the camera position.
EastLimit	Gets or sets the maximum X value allowed for the camera position.
FrontLeftCorner	Gets the point of the front left (southwest) corner of the warehouse.
FrontRightCorner	Gets the point of the front right (southeast) corner of the warehouse.
Height	Gets the height of the warehouse.
Items	Gets a collection of all items in the warehouse.
Length	Gets the length of the warehouse.
NorthLimit	Gets or sets the minimum Z value allowed for the camera position.
SelectedItems	Gets a collection of the currently selected items in the warehouse.
Slots	Gets a collection of the item slots in the warehouse.
SouthLimit	Gets or sets the maximum Z value allowed for the camera position.
UpLimit	Gets or sets the maximum Y value allowed for the camera position.
WestLimit	Gets or sets the minimum X value allowed for the camera position.
Width	Gets the width of the warehouse.

Warehouse Class

Represents a warehouse in the Warehouse3D control.

Syntax

```
public class Warehouse : IWarehouse
```

Constructors

Name	Description
Warehouse(double, double)	Initializes a new instance of the Warehouse class with the specified length and width.
Warehouse(double, double, double)	Initializes a new instance of the Warehouse class with the specified length, width and height.

IWarehouseItem Interface

Represents any type of item that can be contained in the warehouse.

Syntax

```
public interface IWarehouseItem
```

Methods

Name	Description
Deselect	Resets the item to its default color.
Info	The info object to use.
IsSelected	Indicates whether the object is selected or not.
Label	The label to use.
Select	Marks the item as selected in the GUI by changing its color.

WarehouseCylinder Class

A cylinder-shaped warehouse item to use in the Warehouse3D control.

Syntax

```
public class WarehouseCylinder : Cylinder, IWarehouseItem
```

Constructors

Name	Description
WarehouseCylinder(double, double, Point3D, BaseInfoObject)	Initializes a new instance of the WarehouseCylinder class with the specified radius, width, center point of the cylinder side and info object of use.

WarehouseBox Class

A box-shaped warehouse item to use in the Warehouse3D control.

Syntax

```
public class WarehouseBox : Box, IWarehouseItem
```

Constructors

Name	Description
WarehouseBox(double, double, double, Point3D, BaseInfoObject)	Initializes a new instance of the WarehouseBox class with the specified dimensions, center point of the box side and info object of use.

BaseInfoObject Class

The base class for any class holding data about a specific IWarehouseItem object.

Syntax

```
public abstract class BaseInfoObject
```

Constructors

Name	Description
BaseInfoObject(string, int)	Initializes a new instance of a BaseInfoObject class, using a specified ID and layer value.

Properties

Name	Description
ID	Gets or sets the ID of the InfoObject.
Layer	Gets or sets a value indicating the Y-wise layer of the InfoObject.

Example

The following example creates a class that inherits from the BaseInfoObject class and can therefore be used to store information about warehouse items:

```
public class TestInfoObject : BaseInfoObject
{
    public double Price { get; set; }
    public char Quality { get; set; }
    public double Quantity { get; set; }
    public double Weight { get; set; }
    public DateTime Date { get; set; }
    public String Type { get; set; }

    public TestInfoObject(string anID, int aLayer) : base(anID, aLayer) { }
}
```

TextBox3D Class

The base class for any rectangular text field used in the Warehouse3D control.

Syntax

```
public abstract class TextBox3D : ModelVisual3D
```

Constructors

Name	**Description**
TextBox3D(string, Point3D, double)	Initializes a new instance of a TextBox3D class with the specified text, position and width.

Properties

Name	**Description**
Length	Gets or sets the length of the TextBox3D object.
Text	Gets or sets the text to be displayed in the TextBox3D object.
Width	Gets or sets the width of the TextBox3D object.

ItemLabel Class

Represents a glue-on label on a warehouse item.

Syntax

```
public class Warehouse : TextBox3D
```

Constructors

Name	**Description**
ItemLabel(string, Point3D, double, double)	Initializes a new instance of the ItemLabel class with the specified text, position, length, width and the length of its associated warehouse item.
ItemLabel(string, Point3D, double, double, double)	Initializes a new instance of the ItemLabel class with the specified text, position, length of the label side and the length of its associated warehouse item.

Methods

Name	**Description**
Hide	Hides the item label in the GUI.
Show	Shows the item label in the GUI.

ItemSlot Class

Represents an item slot drawn on the warehouse floor.

Syntax

```
public class ItemSlot: TextBox3D
```

Constructors

Name	Description
ItemSlot(string, Point3D, double, double)	Initializes a new instance of the ItemLabel class with the specified text, position, length and width.

Examples of Usage

This example creates a Warehouse3D, adds an item slot and a cylinder-shaped item.

```
 // Create the Warehouse3D.
//There will be a default warehouse within this...
Warehouse3D w = new Warehouse3D();

// ...but we might just as well initialize a new one.
w.Warehouse = new Warehouse(50000,30000);

// Set the initial camera position.
w.CameraPosition = CameraPosition.PerspectiveSouthEast;

// Create an item slot...
Point3D slotPoint = new Point3D(1000, 1, 2000);
ItemSlot mySlot = new ItemSlot("MyItemSlot", slotPoint, 1400, 1500);

// ...and place it in the warehouse.
w.Warehouse.Slots.Add(mySlot);

// Finally, create a cylinder item and place it in the warehouse as well.
TestInfoObject tio = new TestInfoObject("ID", 0);
Point3D cylPoint = new Point3D(1000, 700, 2000);
WarehouseCylinder cyl = new WarehouseCylinder(700, 1300, cylPoint, tio);
w.Warehouse.Items.Add(cyl);
```

Note that this code automatically displays an item label on the warehouse item. This happens within the WarehouseCylinder (or WarehouseBox) constructor, with the ID property in the InfoObject used as the text on the label. In order to display a warehouse item without a label, simply set the Label property to null, like the example below.

Note that this must be done before the warehouse item is added to the model.

```
TestInfoObject tio = new TestInfoObject("ID", 0);
Point3D cylPoint = new Point3D(1000, 700, 2000);
WarehouseCylinder cyl = new WarehouseCylinder(700, 1300, cylPoint, tio);
cyl.Label = null;
w.Warehouse.Items.Add(cyl);
```