SceneryDesign.org

ASToFra Developers Manual

Contents

1 Introduction **1**

2 Conditions for using the libraries **1**

3 Obtaining the libraries **2**

4 Overview of libraries **2**

5 Sample code **3**

5.1 Basic object information 3
5.2 Texture information 4
5.3 Generate footprint of model 6
5.4 Render preview of model 7
5.4.1 Renderer3DControl 10
5.4.2 Load object file 12
5.4.3 Change livery 12
5.4.4 Show statistics 13
5.4.5 Change background color 13

6 Support **13**

7 User license **14**

1 Introduction

ASToFra stands for Arno's Scenery Tool Framework and it is the underlying set of libraries for reading, writing and manipulating Flight Simulator models and sceneries that are used by my tools like ModelConverterX and scenProc. This developers manual explain how other developers can use these libraries in their tools as well, for example because they want to render models or extract information from them. The manual contains sample code on how to do common activities with the libraries. If you have another use case that is not covered by the manual, don't hesitate to contact me.

2 Conditions for using the libraries

My tools that use these libraries are freeware. Therefore developers of other freeware tools are free to use the libraries in their tools as well. I would appreciate it if you let me know when you use my libraries in your tools. That also

allows me to information you about important bug fixes or other changes. Just contact me by email.

If you want to use my libraries in a tool that is not freeware, please contact me beforehand and we can discuss the possibilities.

3 Obtaining the libraries

To obtain the current version of the software libraries, simple download the latest development release of ModelConverterX. You can take the relevant DLL files for your application from ModelConverterX. Your IDE, e.g. Visual Studio, should help you in copying over relevant dependencies of the libraries you want to use, as some of the libraries depend on others. But they are all contained in the ModelConverterX development release.

4 Overview of libraries

The framework consists of multiple libraries, this section gives an overview which functionality is part of which library. You might need different libraries from this list in your application.

- **ASToFra.Coordinates** contains logic related to conversion of coordinates between WGS84 geodetic, round and flat earth.
- **ASToFra.ErrorHandler** contain logic for reporting errors to the Scenery-Design.org bug tracker system.
- **ASToFra.EventLog** contains the controls and logic of the event log as shown at the bottom in ModelConverterX.
- **ASToFra.FSUtils** contains utilities that contain specific logic for FSX and Prepar3D.
- **ASToFra.Geometry** contains the logic that is used to represent vertices, triangles and other geometry.
- **ASToFra.Interface** contains the interfaces that are used between the different libraries.
- **ASToFra.Object.DataModel** contains the logic that is used to represent an object or scenery internally in the tools.
- **ASToFra.Object.Processor** contains the logic to process and modify objects.
- **ASToFra.Object.Reader** contains the logic to read objects and scenery from different formats.
- **ASToFra.Object.Renderer** contains the logic that renders a 3D preview of the object on screen.
- **ASToFra.Object.Writer** contains the logic to write objects and scenery to different formats.

- **ASToFra.ParticleEffects** contains the logic about particle effects as used in FX files.
- **ASToFra.Texture.Object** contains the logic to represent texture information and to search for textures.
- **ASToFra.Texture.Reader** contains the logic to read textures from different formats.
- **ASToFra.Texture.Writer** contains the logic to write textures to different formats.
- **ASToFra.Utils** contains utilities that are used in the other libraries.
- **ASToFra.XPUtils** contains utilities that contain specific logic for X-Plane.

5 Sample code

In this chapter various code samples are given for common activities that can be performed using the libraries.

5.1 Basic object information

This sample command line application loads a library BGL and prints the basic information of all objects in the library to the console.

```
1  using System;
2  using System.Collections.Generic;
3  using ASToFra.FSUtils;
4
5  namespace object_info_basic
6  {
7      public static class Program
8      {
9          public static void Main(string[] args)
10         {
11             BglXStatsReader statisticsReader = new BglXStatsReader();
12
13             List<ObjectStatistics> objectStatistics = statisticsReader.
                   ReadStatistics(args[0]);
14
15             foreach (ObjectStatistics statistics in objectStatistics)
16             {
17                 Console.WriteLine("Name = {0}", statistics.Name);
18                 Console.WriteLine("GUID = {0}", statistics.GUID);
19                 Console.WriteLine("FS version = {0}", statistics.Version)
                       ;
20                 Console.WriteLine("Has animation = {0}", statistics.
                       HasAnimations);
21                 Console.WriteLine("Number of LOD = {0}", statistics.LOD.
                       Count);
22                 Console.WriteLine("Number of textures = {0}", statistics.
                       TextureList.Count);
23                 Console.WriteLine("Bounding box min = {0:0.00}, {1:0.00},
                        {2:0.00}",
24                     statistics.BoundingBox.min.x, statistics.BoundingBox.
                           min.y, statistics.BoundingBox.min.z);
25                 Console.WriteLine("             max = {0:0.00}, {1:0.00},
                        {2:0.00}",
26                     statistics.BoundingBox.max.x, statistics.BoundingBox.
                           max.y, statistics.BoundingBox.max.z);
27                 Console.WriteLine();
28             }
29         }
```

```
    }
}
```

The code first creates an instance of the `BGLXStatsReader` object, this class can be used to retrieve basic statistics of all objects in a BGLComp library BGL file (either FS2004, FSX or Prepar3D). With the `ReadStatistics` a list of `ObjectStatistics` objects is retrieved for the filename that is passed as argument to the console application.

Next this list is looped and for each object that information from the `ObjectStatistics` object is printed to the console. The object does contained more information like the names of the textures and details about the number of drawcalls, vertices and triangles per LOD. But for the simplicity of this example that information is not written to the console.

Figure 1 shows the output this sample program generates.

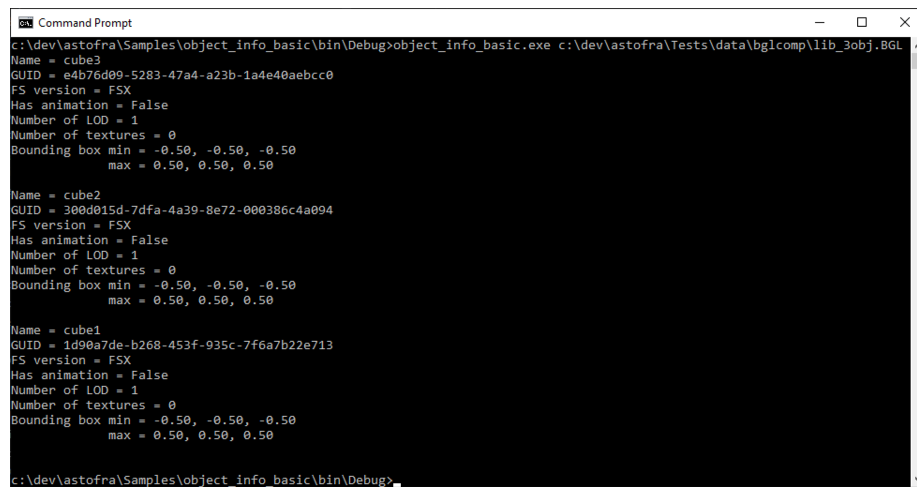

Figure 1: Sample output of basic object information sample project

5.2 Texture information

This sample command line application loads an object and retrieves all textures used by the object and prints the information of these textures to the console.

```
using System;
using ASToFra.Object.Reader;
using ASToFra.Object.DataModel;
using ASToFra.Texture.Object;
using ASToFra.Texture.Reader;
using ASToFra.Interfaces;

namespace texture_info
{
    public static class Program
    {
        public static void Main(string[] args)
        {
            ObjectReaderFactory readerFactory = new ObjectReaderFactory(
                null, null);
```

```
15              AllObjectReader allObjectReader = new AllObjectReader(
                    readerFactory);
16              TextureCache textureCache = new TextureCache(new
                    AllTextureReader(), null, new TextureSearcher());
17
18              Scenery scenery = allObjectReader.Read(args[0]);
19
20              foreach (ObjectModel objectModel in scenery.ObjectModelList)
21              {
22                  Console.WriteLine("Object  | Name = {0}", objectModel.
                        Name);
23
24                  foreach (string texture in objectModel.TextureList)
25                  {
26                      ITextureImage textureImage = textureCache.
                            GetTextureImage(texture, args[0], "");
27
28                      Console.WriteLine("Texture | Name = {0}", texture);
29                      Console.WriteLine("        | Width = {0}",
                            textureImage.Width);
30                      Console.WriteLine("        | Height = {0}",
                            textureImage.Height);
31                      Console.WriteLine("        | Has mipmaps = {0}",
                            textureImage.HasMipMaps);
32                      Console.WriteLine("        | DXT compression = {0}",
                            textureImage.DxtCompression);
33                  }
34
35                  Console.WriteLine("");
36              }
37          }
38      }
39  }
```

So how does this application work?

First an instance of the `ObjectReaderFactor\verbAllObjectReader|` class is created. This class can be used to read an object from any of the formats that is supported. If you want to read from one specific format only, you can also use the reader class that is specific for that format. The `AllObjectReaader` class takes an argument of type `ObjectReaderFactory`, this factory class is used to initialize the specific readers and their post-processing. Since we will only read simple objects from MDL/BGL format here, without the need for post-processing, we can pass `null` arguments to the `ObjectReaderFactory` constructor.

Next an instance of the `TextureCache` class is made. This class can be used to retrieve textures, it does cache the loaded textures to prevent them being read from disk all the time. The constructor takes three arguments:

- An instance of a texture reader that can read the textures into memory. We use an instance of `AllTextureReader` as that class can read any of the formats that are supported by the library.
- An instance of a cube texture reader, since we are not interested in cube textures in this sample we pass `null`.
- An instance of the texture searcher class, this class can find textures on your hard drive using searching logic similar to flight simulator uses.

Then the filename passed as argument to the console application is used to read a `Scenery` object using the `AllObjectReader` instance. And in a loop all `ObjectModel` instances within that scenery are processed. For all the textures in these objects the instance of the `ITextureObject` is retrieved from the texture

cache. If the texture cache does not have this texture stored, it will used to the texture searcher to locate it on disk and then load it with the texture reader instance. The `GetTextureImage` class has 3 arguments, the filename of the texture (without path), object filepath which is used as initial location for searching and optionally the name of the livery to use.

Finally some information about the retrieved texture are printed to the console, like the size of the texture, whether it has mipmaps and which DXT compression is used on the texture.

Figure 2 shows the output this sample program generates.

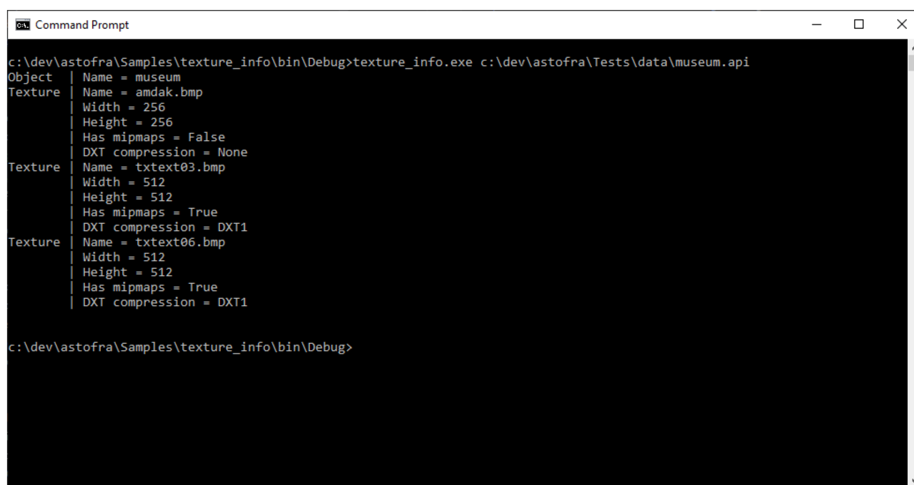

Figure 2: Sample output of texture information sample project

5.3 Generate footprint of model

This sample command line application loads an object, generates the footprint of the object and then prints information about the footprint to the console.

```
1  using System;
2  using System.Collections.Generic;
3  using ASToFra.Geometry;
4  using ASToFra.Object.DataModel;
5  using ASToFra.Object.Processor;
6  using ASToFra.Object.Reader;
7
8  namespace generate_footprint
9  {
10     public static class Program
11     {
12         public static void Main(string[] args)
13         {
14             ObjectReaderFactory readerFactory = new ObjectReaderFactory(
                   null, null);
15             AllObjectReader allObjectReader = new AllObjectReader(
                   readerFactory);
16             FootprintGenerator footprintGenerator = new
                   FootprintGenerator();
17
18             Scenery scenery = allObjectReader.Read(args[0]);
19
20             foreach (ObjectModel objectModel in scenery.ObjectModelList)
21             {
```

```
22                  List<Triangle> footprintTriangles = footprintGenerator.
                        GenerateFootprint(objectModel, 0.1);
23
24                  Console.WriteLine("Object: {0}", objectModel.Name);
25                  foreach (Triangle triangle in footprintTriangles)
26                  {
27                      Console.WriteLine("> [x={0:0.000} y={1:0.000}] [x
                            ={2:0.000} y={3:0.000}] [x={4:0.000} y={5:0.000}]
                            ",
28                          triangle.Vertices[0].Coor.x, triangle.Vertices
                                [0].Coor.y,
29                          triangle.Vertices[1].Coor.x, triangle.Vertices
                                [1].Coor.y,
30                          triangle.Vertices[2].Coor.x, triangle.Vertices
                                [2].Coor.y);
31                  }
32              }
33          }
34      }
35  }
```

Let's walk through the source code in steps.

First an instance of the `AllObjectReader` class is created. This class can be used to read an object from any of the formats that is supported. If you want to read from one specific format only, you can also use the reader class that is specific for that format. As explained in section 5.2 we can pass an instance of the `ObjectReaderFactory` class as argument here that is made using `null` input for the constructor.

Next an instance of the `FootprintGenerator` class is made. This is the class that can calculate the footprint from an object.

Then we load the scenery object from the file that is passed as argument to the command prompt application. And we iterate over all `ObjectModel` instances in the scenery to calculate the footprint for each of them. The `GenerateFootprint` function takes the `ObjectModel` instance as argument and you can also specify the minimum area (in square meters) that geometry should have to be included in the footprint. In this example it has been set to 0.1 square meter, but if you want to reduce the complexity of the footprint you can increase this value as well.

Finally the X and Y position of each triangle in the footprint is printed to the console output so that you can see the footprint that was generated. If you would use the footprint in your own tool, you would probably render the triangles to your screen or use the information otherwise.

Figure 3 shows the output this sample program generates.

5.4 Render preview of model

In this example a more complex application is discussed. It is an application that uses the preview control of ModelConverterX to be able to visualize 3D objects. Figure 4 shows how this application looks like.

The application is created using the `Renderer3DControl` control. When you place this control in your WinForms application you will get a fully functioning 3D preview. Four buttons have been placed on the form as well that are used to show how you can interact with the preview control or with the object that

Command Prompt

```
c:\dev\astofra\Samples\generate_footprint\bin\Debug>generate_footprint.exe c:\dev\astofra\Tests\data\museum.api
Object: museum
> [x=-0.400 y=11.200] [x=-0.400 y=-11.100] [x=0.100 y=11.200]
> [x=-0.400 y=-11.100] [x=0.100 y=-11.100] [x=0.100 y=11.200]
> [x=8.000 y=11.200] [x=0.000 y=11.200] [x=8.000 y=-11.100]
> [x=0.000 y=11.200] [x=0.000 y=-11.100] [x=8.000 y=-11.100]
> [x=8.000 y=-11.100] [x=16.000 y=-11.100] [x=8.000 y=11.200]
> [x=16.000 y=-11.100] [x=16.000 y=11.200] [x=8.000 y=11.200]
> [x=-4.400 y=11.200] [x=-8.400 y=11.200] [x=-4.400 y=-11.100]
> [x=-8.400 y=11.200] [x=-8.400 y=-11.100] [x=-4.400 y=-11.100]
> [x=-4.400 y=-11.100] [x=-0.400 y=-11.100] [x=-4.400 y=11.200]
> [x=-0.400 y=-11.100] [x=-0.400 y=11.200] [x=-4.400 y=11.200]
> [x=-6.900 y=-11.100] [x=-6.900 y=-12.600] [x=-3.900 y=-11.100]
> [x=-6.900 y=-12.600] [x=-3.900 y=-12.600] [x=-3.900 y=-11.100]

c:\dev\astofra\Samples\generate_footprint\bin\Debug>
```

Figure 3: Sample output of generate footprint sample project

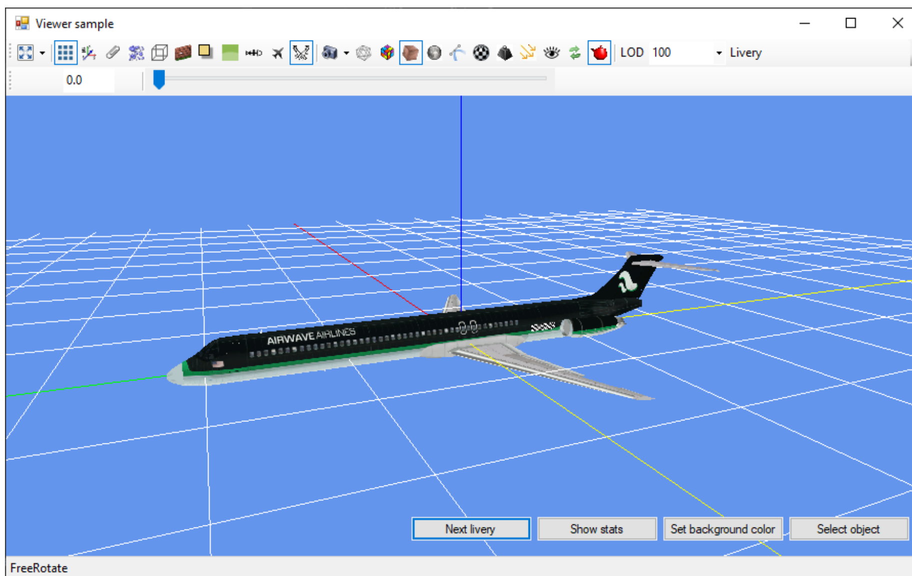

Figure 4: Example application using the 3D preview of ModelConverterX

is being shown. Below the full source code of this application is shown, but in the sections below we will discuss a different part of the source code at a time.

```
1  using System;
2  using System.Windows.Forms;
3  using ASToFra.Object.Reader;
4  using ASToFra.Object.Renderer;
5  using ASToFra.Object.Processor;
6  using ASToFra.Interfaces;
7  using ASToFra.Object.DataModel;
8  using System.IO;
9  using ASToFra.FSUtils;
10 using ASToFra.Texture.Object;
11 using ASToFra.Texture.Reader;
12 using ASToFra.Object.DataModel.Interfaces;
13
14 namespace viewer_sample
15 {
16     public partial class Form1 : Form, IReceiveObjectChanged,
           ITransmitObjectChanged
17     {
18         public event ObjectChanged ObjectChanged;
19
20         readonly ITextureCache _textureCache;
21         readonly AllObjectReader _objectReader;
22         ObjectModel _currentObject;
23
24         public Form1()
25         {
26             InitializeComponent();
27
28             _textureCache = new TextureCache(new AllTextureReader(), new
                   DdsLoader(), new TextureSearcher());
29             renderer3DControl1.InitializeHandlers(_textureCache, null,
                   new RendererState());
30
31             // Passing null here for the textureWriter as I now that the
                   reading logic does not use it.
32             IObjectProcessorFactory processFactory = new
                   ObjectProcessorFactory(_textureCache, null);
33             ObjectReaderFactory readerFactory = new ObjectReaderFactory(
                   processFactory, _textureCache);
34             _objectReader = new AllObjectReader(readerFactory);
35
36             FsUtilsSettingsWrapper fsSetting = new FsUtilsSettingsWrapper
                   ();
37             fsSetting.AutoDetectSettings();
38
39             EventDistributor.Instance.RegisterComponent(this);
40         }
41
42         private void Form1_FormClosing(object sender,
               FormClosingEventArgs e)
43         {
44             EventDistributor.Instance.UnRegisterComponent(this);
45         }
46
47         public void OnObjectChanged(ObjectChangeLevel level, object obj,
               object sender, string changeDescription)
48         {
49             _currentObject = (ObjectModel) obj;
50         }
51
52         private void RaiseObjectChanged()
53         {
54             if (ObjectChanged != null)
55             {
56                 ObjectChanged(ObjectChangeLevel.
                       MaterialChangedTextureReload, _currentObject, this, "
                       New livery");
57             }
58         }
59
```

```
60          private void cmdSelect_Click(object sender, EventArgs e)
61          {
62              OpenFileDialog openFile = new OpenFileDialog();
63              if (openFile.ShowDialog() == DialogResult.OK)
64              {
65                  _objectReader.ReadASync(openFile.FileName);
66              }
67          }
68
69          private void cmdColor_Click(object sender, EventArgs e)
70          {
71              ColorDialog color = new ColorDialog();
72              if (color.ShowDialog() == DialogResult.OK)
73              {
74                  Renderer3DSettingsWrapper settings = new
                        Renderer3DSettingsWrapper();
75                  settings.BackgroundColor = color.Color;
76                  renderer3DControl1.Refresh();
77              }
78          }
79
80          private void cmdLivery_Click(object sender, EventArgs e)
81          {
82              if (_currentObject != null && _currentObject.Liveries.Count >
                     1)
83              {
84                  int index = _currentObject.ActiveLiveryIndex + 1;
85                  if (index >= _currentObject.Liveries.Count)
86                  {
87                      index = 0;
88                  }
89
90                  _currentObject.ActiveLiveryIndex = index;
91                  RaiseObjectChanged();
92              }
93          }
94
95          private void cmdStats_Click(object sender, EventArgs e)
96          {
97              if (_currentObject != null)
98              {
99                  ObjectStatistics stats = _currentObject.GetStatistics();
100                 StatisticsLod lod = stats.LOD[stats.LOD.Count - 1];
101                 StringWriter writer = new StringWriter();
102                 writer.WriteLine("LOD: {0}", lod.Value);
103                 writer.WriteLine("Triangles: {0}", lod.Triangles);
104                 writer.WriteLine("Vertices: {0}", lod.Vertices);
105                 MessageBox.Show(writer.ToString());
106             }
107         }
108     }
109 }
```

5.4.1 Renderer3DControl

When you insert the `Renderer3DControl` in your WinForms application you basically have a functioning preview automatically. When you load an object using the `AllObjectReader` the preview control will respond to the event that is raised when the object has finished loading and show this object in the preview.

For the `Renderer3DControl` instance to work correctly you need to call the `InitializeHandlers` function directly after the instance has been created, e.g. in the constructor of your form. This function takes three arguments:

1. A `TextureCache` object that is used by the preview to retrieve the object textures from. The instance of the `TextureCache` is made in the construc-

tor of the application and passed to the InitializeHandlers function.

2. An ErrorHandler object that is used to report errors, since reporting errors is disabled in this sample we pass null.

3. An RendererState object that is used by the preview to retrieve the state of visibility conditions from. We create a new instance of this class to pass to the function.

In the constructor of the form we also create instances of some helper class that the control uses. The most import one is the AllObjectReader class, as this is the class that reads an object model into memory so that the preview can display it. As the AllObjectReader needs an instance of the ObjectProcessorFactory class as well, we create an instance of that class as well. This factory class is used for the post-processing after the object has been loaded. It for example makes sure that textures with an alpha channel are setup correctly. The ObjectProcessorFactory constructor takes two arguments, the first is the TextureCache that we already created for the preview control itself. The second argument is for a TextureWriter, but since we don't need to write textures for the preview we pass a null for that one.

To make sure that effect file in the FS effects folder are found correctly we also let the application detect where FS is installed. This is done by calling the AutoDetectSettings function on the |FSUtilsSettingsWrapper| class.

Finally we need to register our form with hte EventDistributor class. This makes sure we can send and receive events with others. This is used in some actions later on where we have updated the object and want to inform the preview control about it. The form also contains two helper functions to work with these events.

The first helper function receives an event once the object has changed, for example when the reader has finished reading the object. It stores the ObjectModel instance of the new object in a variable, so that we can use it in other functions.

```
1 public void OnObjectChanged(ObjectChangeLevel level, object obj, object
      sender, string changeDescription)
2 {
3     _currentObject = (ObjectModel) obj;
4 }
```

The second helper function raises an event once we have modified the object ourself. In this case the ObjectChangeLevel is set to MaterialChangedTextureReload since we will use this function when we change the livery later on. The preview control will be trigged by this event to reload its textures.

```
1 private void RaiseObjectChanged()
2 {
3     if (ObjectChanged != null)
4     {
5         ObjectChanged(ObjectChangeLevel.MaterialChangedTextureReload,
              _currentObject, this, "New livery");
6     }
7 }
```

The `Renderer3DControl` also has some properties that you can set from the IDE. These control how the control itself behaves. The most relevant properties are:

- **ShowToolbar** controls if a toolbar is shown at the top. In this toolbar there are buttons to toggle if normals, a grid, etc. are shown. If you disable the toolbar the users can not modify how the object is rendered by themselves. Unless you modify these properties from your source code.
- **ShowStatusbar** controls if a statusbar is shown at the bottom. In this statusbar you can see the rotation mode that is selected and information while an object is loaded into the preview.

5.4.2 Load object file

Once you have added the `Renderer3DControl` to your application one of the first things you want to do is to load an object into the preview.

With all the class that we need instantiated, all we have to do is to select the file we want to load and ask the `AllObjectReader` to load this file. The reader will automatically generate the event that triggers the preview to update itself. Below is the code that shows the file selection dialog and then ask the reader to load that file asynchronous.

```
1 private void cmdSelect_Click(object sender, EventArgs e)
2 {
3     OpenFileDialog openFile = new OpenFileDialog();
4     if (openFile.ShowDialog() == DialogResult.OK)
5     {
6         _objectReader.ReadASync(openFile.FileName);
7     }
8 }
```

5.4.3 Change livery

With the Next livery button on the form you can tell the preview that it should show the next livery for the loaded object. This will only work if you have loaded an aircraft model that contains multiple liveries of course. Below is the source code behind this button.

```
1 private void cmdLivery_Click(object sender, EventArgs e)
2 {
3     if (_currentObject != null && _currentObject.Liveries.Count > 1)
4     {
5         int index = _currentObject.ActiveLiveryIndex + 1;
6         if (index >= _currentObject.Liveries.Count)
7         {
8             index = 0;
9         }
10
11        _currentObject.ActiveLiveryIndex = index;
12        RaiseObjectChanged();
13    }
14 }
```

So what happens? First it is checked if we have an object loaded and if that object has multiple liveries. If that is the case the index of the active livery is increased by one. If the new index is greater or equal to the number of liveries, we reset to index to zero. This way we cycle through all liveries. This new

livery index is set on the object and then we raise an event that we changed the object. This will force the preview to update the livery.

5.4.4 Show statistics

When you click the Show statistics button some information about the loaded object is shown in a message box. This is similar in function to the console application that was discussed before, but in this case it is integrated in the forms. It will show the number of vertices and triangles in the highest level of detail of the model.

```
1 private void cmdStats_Click(object sender, EventArgs e)
2 {
3     if (_currentObject != null)
4     {
5         ObjectStatistics stats = _currentObject.GetStatistics();
6         StatisticsLOD lod = stats.LOD[stats.LOD.Count - 1];
7         StringWriter writer = new StringWriter();
8         writer.WriteLine("LOD: {0}", lod.Value);
9         writer.WriteLine("Triangles: {0}", lod.Triangles);
10        writer.WriteLine("Vertices: {0}", lod.Vertices);
11        MessageBox.Show(writer.ToString());
12    }
13 }
```

5.4.5 Change background color

With the Set background color button you can change the color that is used in the preview for the background. All you have to do is select a new color, we use the default `ColorDialog` for that and then pass the new color to the correct field of the `Renderer3DSettingsWrapper` class. In this case we set the `BackgroundColor` field, but you can also configure many other colors.

```
1 private void cmdColor_Click(object sender, EventArgs e)
2 {
3     ColorDialog color = new ColorDialog();
4     if (color.ShowDialog() == DialogResult.OK)
5     {
6         Renderer3DSettingsWrapper settings = new
              Renderer3DSettingsWrapper();
7         settings.BackgroundColor = color.Color;
8         renderer3DControl1.Refresh();
9     }
10 }
```

In this way you can also change other properties of the preview control, as there are also fields in the `Renderer3DSettingsWrapper` class that for example control if the grid is shown.

6 Support

If you need help in using these libraries in your tool, if the functionality you want to use is not explained in this manual or if you are missing certain features for you tool, please contact me by email.

7 User license

(c) 2007-2024 SceneryDesign.org / Arno Gerretsen

These software libraries are distributed without charge to other addon developers. Redistribution of the original DLL as part of your tool without charge is allowed. You are NOT allowed to sell these software libraries itself or ask money for its distribution.

The copyright and any intellectual property relating to this program remain the property of the author.

The software distributed in this way may represent work in progress, and bears no warranty, either expressed or implied.