

HOpenGL – 3D Graphics with Haskell

A small Tutorial

(Draft)

Version 9th October 2003

Sven Eric Panitz
TFH Berlin

Publish early and publish often. That is the reason why you can read this. I started playing around with *HOpenGL* the Haskell port of OpenGL a common library for doing 3D graphics. I more or less took minutes of my efforts and make them public in this tutorial. I did not have any prior experience in graphics programming, when I started to work with HOpenGL.

The source of this paper is an XML-file. The sources are processed by an XQuery processor, XSLT scripts and \LaTeX in order to produce the different formats of the tutorial.

I'd like to thank Sven Panne¹, the author of HOpenGL, who has been so kind to comment on first drafts of this tutorial.

¹Similar name different person.

Contents

1	Introduction	1-1
1.1	A Little Bit of Practice	1-1
1.1.1	Opening Windows	1-1
1.1.2	Drawing into Windows	1-2
1.2	A Little Bit of Theory	1-5
1.2.1	Haskell	1-5
1.2.2	OpenGL	1-5
1.2.3	Haskell and OpenGL	1-5
1.3	A Little Bit of Technics	1-6
1.4	A Little Bit of History	1-6
2	Basics	2-1
2.1	Setting and Getting of Variables	2-1
2.1.1	Setting values	2-1
2.1.2	Getting values	2-2
2.1.3	Getting and Setting Values	2-3
2.1.4	What do the variables refer to	2-4
2.2	Basic Drawing	2-4
2.2.1	Display Functions	2-4
2.2.2	Primitive Shapes	2-6
2.2.3	Curves, Circles and so on	2-16
2.2.4	Attributes of primitives	2-20
2.2.5	Tessellation	2-24
2.2.6	Cubes, Dodecahedrons and Teapots	2-26

3	Modelling Transformations	3-1
3.1	Translate	3-1
3.2	Rotate	3-2
3.3	Scaling	3-4
3.4	Composition of Transformations	3-5
3.5	Defining your own transformation	3-8
3.5.1	Shear	3-9
3.6	Some Word of Warning	3-10
3.7	Local transformations	3-11
4	Projection	4-1
4.1	The Function Reshape	4-1
4.2	Viewport: The Visible Part of Screen	4-2
4.3	Orthographic Projection	4-4
5	Changing States	5-1
5.1	Modelling your own State	5-1
5.2	Handling of Events	5-2
5.2.1	Keyboard events	5-2
5.3	Changing State over Time	5-4
5.3.1	Double buffering	5-5
5.4	Pong: A first Game	5-6
6	Third Dimension	6-1
6.1	Hidden Shapes	6-1
6.2	Perspective Projection	6-4
6.3	Setting up the Point of View	6-7
6.3.1	Oribiting around the origin	6-8
6.4	3D Game: Rubik's Cube	6-11
6.4.1	Cube Logics	6-12
6.4.2	Rendering the Cube	6-14
6.4.3	Rubik's Cube	6-16
6.5	Light	6-17
6.5.1	Defining a light source	6-19
6.5.2	Tux the Penguin	6-20
	Haskell Examples	30

Chapter 1

Introduction

In this chapter some basic background information can be found. You can read the sections of this chapter in an arbitrary order. Whatever your personal preference is.

1.1 A Little Bit of Practice

Before you read a lot of technical details you will probably like to see something on your screen. Therefore you find some very simple examples in the beginning. This will give you a first impression, of how an OpenGL program might look like in Haskell.

1.1.1 Opening Windows

OpenGL's main purpose is to render some graphics on a device. This device is generally a window on your computer screen. Before you can draw something on a screen you will need to open a window. So let's have a look at the simplest OpenGL program, which just opens an empty window:

```
----- HelloWorld.hs -----
1 import Graphics.UI.GLUT
2 import Graphics.Rendering.OpenGL
3
4 main = do
5     getArgsAndInitialize
6     createAWindow "Hello Window"
7     mainLoop
8
9 createAWindow windowName = do
10     createWindow windowName
11     displayCallback $= clear [ColorBuffer]
```

The first two lines import the necessary libraries. The main function does three things:

- initialize the OpenGL system

- define a window
- start the main procedure for displaying everything and reacting on events

For the definition of a window with a given name we do two things:

- create some window with the given name
- define, what is to be done, when the window contents is to be displayed. In the simple example above we simply clear the screen of any color by filling it with the default background color.

This 10 lines can be compiled with `ghc`. Do not forget to specify the packages, which contain the OpenGL library. It suffices to include the package `GLUT`, which automatically forces the inclusion of the package `OpenGL`. GLUT is the graphical user interface, which comes along with OpenGL, i.e. the window managing system etc.

```
sep@swe10:~/hopengl/examples> ghc -package GLUT -o HelloWorld HelloWorld.hs
sep@swe10:~/hopengl/examples> ./HelloWindow
```

When you start the program, a window will be opened on your desktop. As you may have noticed, we did not specify any attribute of the window, like its size and position. GLUT is defined in a way that initial default values are used for unspecified attributes.

1.1.2 Drawing into Windows

The simple program above did just open a window. The main purpose of OpenGL is to define some graphics which is rendered in a window. Before starting to systematically explore the OpenGL library let's have a look at two examples that draw something into a window frame.

Some Points

First we will draw some tiny points on the screen. We use the same code for opening some window:

```

----- SomePoints.hs -----
1 import Graphics.UI.GLUT
2 import Graphics.Rendering.OpenGL
3
4 main = do
5     (progName,_) <- getArgsAndInitialize
6     createAWindow progName
7     mainLoop
```

The only thing that has changed, is that we make use of one of the values returned by `getArgsAndInitialize`: the name of the program.

For the window definition we use the code from `HelloWindow.hs`. But instead of clearing the screen, when the window is to be displayed, we use an own display function:

```

10 createAWindow windowName = do
11     createWindow windowName
12     displayCallback $= displayPoints

```

We want to draw some points on the screen. So let's define some points. We can do this in a list. Points in a three dimensional space are triples of coordinates. We can use floating point numbers for coordinates in OpenGL.

```

11 myPoints :: [(GLfloat,GLfloat,GLfloat)]
12 myPoints =
13     [(-0.25, 0.25, 0.0)
14      ,(0.75, 0.35, 0.0)
15      ,(0.75, -0.15, 0.0)
16      ,((-0.75), -0.25, 0.0)]

```

Eventually we need the display function, which displays these points.

```

17 displayPoints = do
18     clear [ColorBuffer]
19     renderPrimitive Points
20     $mapM_ \(x, y, z)->vertex$Vertex3 x y z) myPoints

```

As you see, when the window is displayed, we want first everything to be cleared from the window. Then we use the HOpenGL function `renderPrimitive`. The first argument `Point` specifies what it is that we want to render; points in our case. For the second argument we need to transform our coordinates into some data, which is used by HOpenGL. Do not yet worry about this transformation.

As before, you will notice that again for quite a number of attributes we did not supply explicit values. We did not specify the Color of the points to be drawn. Moreover we did not define the coordinates of the graphics window. Looking at its result it is obviously a two dimensional view, where the lower left corner seems to have coordinates (-1,-1) and the upper right corner the (1,1). These values are default values chosen by the OpenGL library.

A Polygon

The points in the last section were rather boring? By changing a single word, we can span an area with these points. Instead of saying render the following as points, we can tell HOpenGL to render them as a polygon.

Beispiel:

So here the program from above with one word changed. `Points` becomes `Polygon`.

```
APolygon.hs
1 import Graphics.UI.GLUT
2 import Graphics.Rendering.OpenGL
3
4 main = do
5     (progName,_) <-getArgsAndInitialize
6     createAWindow progName
7     mainLoop
8
9 createAWindow windowName = do
10     createWindow windowName
11     displayCallback $= displayPoints
12
13 displayPoints = do
14     clear [ColorBuffer]
15     renderPrimitive Polygon
16     $mapM_ (\(x, y, z)->vertex$Vertex3 x y z) myPoints
17
18 myPoints :: [(GLfloat,GLfloat,GLfloat)]
19 myPoints =
20     [(-0.25, 0.25, 0.0)
21     ,(0.75, 0.35, 0.0)
22     ,(0.75, -0.15, 0.0)
23     ,((-0.75), -0.25, 0.0)]
```

The resulting window can be found in figure 1.1.

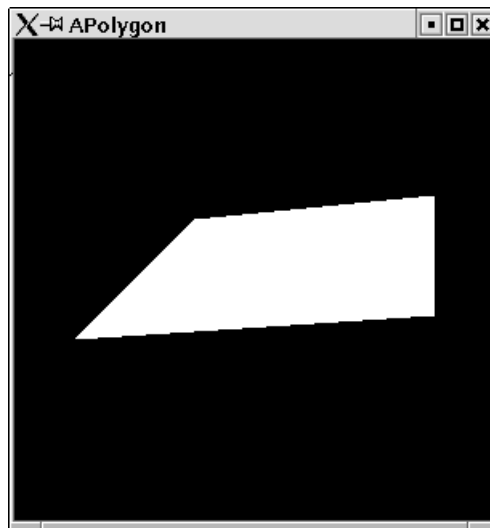


Figure 1.1: A simple polygon.

1.2 A Little Bit of Theory

1.2.1 Haskell

Haskell [Jon03] is a lazily evaluated functional programming language. This means that there are no mutable variables. A Haskell program consists of expressions, which do not have any side effects. Expressions are only evaluated to some value when this is absolutely necessary for program execution. This means it is hard to predict in which order subexpressions get evaluated.

Expressions evaluate to some value without changing any state. This is a nice property of Haskell, because it makes reasoning about programs easier and programs are very robust.

1.2.2 OpenGL

OpenGL on the other hand is a graphics library which is defined in terms of a state machine. A mutable state models the current state of the world. Functions are executed one after another on this state in order to modify certain variables. E.g. one variable keeps the current color to which all drawing statements refer. There is a statement which allows to set the color variable to some other value.

A comprehensive introduction to OpenGL can be found in the so called *redbook* [WBN⁺97]. OpenGL comes along with a utility library called GLU [CFH⁺98] and a system independent GUI library called GLUT [Kil96].

1.2.3 Haskell and OpenGL

Having said this, Haskell and OpenGL seem to cooperate badly. There seems to be a great mismatch between the fundamental concepts of the two. However, the designers of Haskell discovered a very powerful structure, which is a perfect concept for modelling state changing functions in a purely functional language: *Monads* [Wad90]. Most Haskell programmers do not worry about the theory of monads but simply use them, whenever they do I/O, state changing functions or in parser construction. With monads functional programs can almost look like ordinary imperative programs [PJW93].

Monads are so essential to functional programming, that they have a special syntactic construct in Haskell, the *do notation*.

Consider the following simple Haskell program, which uses monads:

```
Print.hs
1  main = do
2      let x = 5
3      print x
4      let x = 6
5      print x
6      xs  <- getLine
7      print (length xs)
```


The monadic statements start with the keyword `do`. The statements have side effects. Variables can be defined and redefined in `let`-expressions¹. Monadic statements can have a result. This can be retrieved from the statement by the `<-` notation.

On another aspect OpenGL and Haskell perfectly match. In OpenGL functions are assigned to different data objects, e.g. a display function is passed to windows. Since functions are first class citizens, they can easily and type safe be passed around².

1.3 A Little Bit of Technics

If you want to start programming OpenGL in Haskell you need to be one of the brave, who compile sources from the functional programming CVS repository in Glasgow. There is not yet a precompiled version of the current HOpenGL library. Go to the website (www.haskell.org/ghc) of the Glasgow Haskell Compiler (GHC), follow closely the instructions on the page *CVS cheat sheet*. When doing the `./configure` step, then use the option `--enable-hopengl`. i.e. start the command `./configure --enable-hopengl`. This will ensure that the Haskell OpenGL library will be build and the packages OpenGL and GLUT are added to your GHC installation.

To compile Haskell OpenGL programs you simply have to add the package information to the command line invocation of GHC, i.e. use:

```
ghc -package GLUT MyProgram.hs
```

Everything else, linking etc is done by GHC. You do not have to worry about library paths or anything else.

1.4 A Little Bit of History

The Haskell port of OpenGL has been done by Sven Panne. Currently a stable version exists and can be downloaded as precompiled binary. This tutorial deals with the completely revised version of HOpenGL, which has a more Haskell like API and needs less technical overhead. This new version is not yet available as ready to use package. You need to compile it yourself.

This tutorial has been written with no prior knowledge of OpenGL and no documentation of HOpenGL at hand.

For the *old* version 1.04 of HOpenGL an online tutorial written by Andre W B Furtado exists at (www.cin.ufpe.br/~haskell/hopengl/index.html) .

¹Variables bound in `let`-expressions are not variables as known from imperative languages. Line 4 in the example above does not assign a new value to a variable `x` but defines a new variable `x`.

²Unlike the object orientated languages Java, which misses an easy way to pass functions around.

Chapter 2

Basics

2.1 Setting and Getting of Variables

From what we have learnt in the introduction, we know that we are dealing with a state machine and will write a sequence of monadic functions which effect this machine. Before we start drawing fancy pictures let us explore the way values are set and retrieved in HOpenGL.

2.1.1 Setting values

The most basic operation is to assign values to variables in the state machine. In HOpenGL this is done by means of the operator `$=`¹. You do not need to understand, how this operator is implemented. You simply can imagine that it is an assignment operator. The left operand is a variable which gets assigned the right operand. We can revisit the first program, which simply opened a window.

Beispiel:

When we have created a window, we assign a size to it:

```
Set.hs
1 import Graphics.UI.GLUT
2 import Graphics.Rendering.OpenGL
3
4 main = do
5     getArgsAndInitialize
6     myWindow "Hello Window"
7     mainLoop
8
9 myWindow name = do
10     createWindow name
11     windowSize $= Size 800 500
12     displayCallback $= clear [ColorBuffer]
```

¹A nicer choice for this operator would have been `:=`, but this is not allowed for a function operator in Haskell, but denotes an infix constructor.

One example of the assignment operator we have already seen. In the last line we assign a function to the variable `displayCallback`. This function will be executed, whenever the window is displayed.

As you see, more you do not need to know about `$=`. But if you want to learn more about it read the next section.

Implementation of set

The operator `$=` is defined in the module `Graphics.Rendering.OpenGL.GL.StateVar` as a member function of a type class:

```

1 infixr 2 $=
2
3 class HasSetter s where
4   ($=) :: s a -> a -> IO ()

```

The variables of `HOpenGL`, which can be set are of type `SettableStateVar` e.g.: `windowTitle :: SettableStateVar String`. Further variables that can be set for windows are: `windowStatus`, `windowTitle`, `iconTitle`, `pointerPosition`,

2.1.2 Getting values

You might want to retrieve certain values from the state. This can be done with the function `get`, which is in a way the corresponding function to the operator `$=`.

Beispiel:

You can retrieve the size of the screen:

```

1 import Graphics.UI.GLUT
2 import Graphics.Rendering.OpenGL
3
4 main = do
5   getArgsAndInitialize
6   x<-get screenSize
7   print x

```

When you compile and run this example the size of your screen it printed:

```

sep@swe10:~/hopengl/examples> ghc -package GLUT -o Get Get.hs
sep@swe10:~/hopengl/examples> ./Get
Size 1024 768
sep@swe10:~/hopengl/examples>

```

Implementation of get

There is a corresponding type class, which denotes that values can be retrieved from a variable:

```
1 class HasGetter g where
2   get :: g a -> IO a
```

Variables which implement this class are of type `GettableStateVar a`.

2.1.3 Getting and Setting Values

For most variables you would want to do both: setting them and retrieving their values. These variables implement both type classes and are usually of type: `StateVar`.

But things do not always work so simple as this sounds.

Beispiel:

The following program sets the size of a window. Afterwards the variable `windowSize` is retrieved:

```
----- SetGet.hs -----
1 import Graphics.UI.GLUT
2 import Graphics.Rendering.OpenGL
3
4 main = do
5   getArgsAndInitialize
6   myWindow "Hello Window"
7   mainLoop
8
9 myWindow name = do
10   createWindow name
11   windowSize $= Size 800 500
12   x<-get windowSize
13   print x
14   displayCallback $= clear [ColorBuffer]
```

Running this program gives the somehow surprising result:

```
sep@swe10:~/hopengl/examples> ./SetGet
Size 300 300
```

The window we created, has the expected size of (800,500) but the variable `windowSize` still has the default value (300,300).

The reason for this is, that setting the window size state variable has not a direct effect. It just states a wish for a window size. Only in the execution of the function `mainLoop` actual windows will be created by the window system. Only then the window size will be taken into account. Up to that moment the window size variable still has the default value. If you print the window size state within some function which is executed in the main loop, then you will get the actual size. By the way: you can try `initialWindowSize` without getting such complicated surprising results.

2.1.4 What do the variables refer to

The state machine contains variables and stacks of objects, which are effectedly mutated by calls to monadic functions. However not only the `get` and `set` statements modify the state but also statements like `createWindow`. This makes it in the beginning a bit hard to understand, when the state is changed in which way.

The `createWindow` statement not only constructs a window object, but keeps this new window as the current window in the state. After the `createWindow` statement all window effecting statements like setting the window size, are applied to this new window object.

2.2 Basic Drawing

2.2.1 Display Functions

There is a window specific variable which stores the function that is to be executed whenever a window is to be displayed, the variable `displayCallback`. Since Haskell is a higher order language, it is very natural to pass a function to the assignment operator. We can define a function with some arbitrary name. The function can be assigned to the variable `displayCallback`. In this function we can define a sequence of monadic statements.

Clearing the Screen

A first step we would like to do whenever the window needs to be drawn is to clear from it whatever it contains². `HOpenGL` provides the function `clear`, which does exactly this job. It has one argument. It is a list of objects to be cleared. Generally you will clear the so called color buffer, which contains the color displayed for every pixel on the screen.

Beispiel:

The following simple program opens a window and clears its content pane whenever it is displayed:

```
1  import Graphics.UI.GLUT
2  import Graphics.Rendering.OpenGL
3
4  main = do
5      (progName,_) <- getArgsAndInitialize
6      createAWindow progName
7      mainLoop
8
9  createAWindow windowName = do
10     createWindow windowName
11     displayCallback $= display
12
13  display = clear [ColorBuffer]
```

²Otherwise you might see arbitrary parts of other applications in your window frame.

First Color Operations

The window in the last section has a black background. This is because we did not specify the color of the background and HOpenGL's default value for the background color is black. There is simply a variable for the background color.

For colors several data types are defined. An easy to use one is:

```
1 data Color4 a = Color4 a a a a
2   deriving ( Eq, Ord, Show )
```

The four parameters of this constructor specify the red, green and blue values of the color and additionally a fourth argument, which denotes the opaqueness of the color. The values are usually specified by floating numbers of type `GLfloat`. Values for number attributes are between 0 and 1.

You may wonder, why there is a special type `GLfloat` for numbers in HOpenGL. The reason is that OpenGL is defined in a way that it is as independent from concrete types in any implementation as possible. However you do not have to worry too much about this type. You can use ordinary float literals for numbers of type `GLfloat`. Haskell's overloading mechanism ensures that these literals can create `GLfloat` numbers.

Beispiel:

This program opens a window with a red background.

```

                                BackgroundColor.hs
1  import Graphics.UI.GLUT
2  import Graphics.Rendering.OpenGL
3
4  main = do
5      getArgsAndInitialize
6      createAWindow "red"
7      mainLoop
8
9  createAWindow windowName = do
10     createWindow windowName
11     displayCallback $= display
12
13  display = do
14     clearColor $= Color4 1 0 0 1
15     clear [ColorBuffer]
```

Committing Complete Drawing

Whenever in a display function a sequence of monadic statements is defined, a final call to the function `flush` should be made. Only such a call will ensure that the statements are completely committed to the device, on which is drawn.

2.2.2 Primitive Shapes

So most preparatory things we know by now. We can start drawing onto the screen. Astonishingly in OpenGL there is only very limited number of shapes for drawing. Just points, simple lines and polygons. No curves or more complicated objects. Everything needs to be performed with these primitive drawing functions. The main function used for drawing something is `renderPrimitive`. The first argument of this functions specifies what kind of primitive is to be drawn. There are the following primitives defined in OpenGL:

```

1 data PrimitiveMode =
2     Points
3     | Lines
4     | LineLoop
5     | LineStrip
6     | Triangles
7     | TriangleStrip
8     | TriangleFan
9     | Quads
10    | QuadStrip
11    | Polygon
12    deriving ( Eq, Ord, Show )

```

The second argument defines the points which specify the primitives. These points are so called vertexes. Vertexes are actually monadic functions which constitute a point. If you want to define a point in a 3-dimensional universe with the coordinates x, y, z then you can use the following expression in HOpenGL:

```
vertex (Vertex3 x y z)
```

or, if you prefer the use of the standard prelude operator `$`:

```
vertex$Vertex3 x y z
```

Points

We have seen in the introductory example that we can draw points. We can simply define a vertex and use this in the function `renderPrimitive`.

Beispiel:

This program draws one single yellow point on a black screen.

```

_____ SinglePoints.hs _____
1 import Graphics.UI.GLUT
2 import Graphics.Rendering.OpenGL
3
4 main = do
5     getArgsAndInitialize
6     createAWindow "points"

```

```

7   mainLoop
8
9   createAWindow windowName = do
10    createWindow windowName
11    displayCallback $= display
12
13   display = do
14    clear [ColorBuffer]
15    currentColor $= Color4 1 1 0 1
16    renderPrimitive Points
17    (vertex (Vertex3 (0.1::GLfloat) 0.5 0))
18    flush

```

If you do not like parantheses then you can of course use the operator `$` from the prelude and rewrite the line:

```
renderPrimitive Points$vertex$Vertex3 (0.1::GLfloat) 0.5 0
```

Unfortunately Haskell needs sometimes a little bit of help for overloaded type classes. Therefore you find the type annotation `(0.1::GLfloat)` on one of the float literals. In larger applications Haskell can usually infer this information from the context. Just in smaller applications you will sometimes need to help Haskell's type checker a bit.

The second argument of `renderPrimitive` is a sequence of monadic statements. So, if you want more than one point to be drawn, you can define these in a nested *do statement*

Beispiel:

In this program we use a nested *do statement* to define more points.

```

MorePoints.hs
1   import Graphics.UI.GLUT
2   import Graphics.Rendering.OpenGL
3
4   main = do
5     getArgsAndInitialize
6     createAWindow "more points"
7     mainLoop
8
9   createAWindow windowName = do
10    createWindow windowName
11    displayCallback $= display
12
13   display = do
14    clear [ColorBuffer]
15    currentColor $= Color4 1 1 0 1
16    renderPrimitive Points $
17    do
18      vertex (Vertex3 (0.1::GLfloat) 0.6 0)
19      vertex (Vertex3 (0.1::GLfloat) 0.1 0)
20    flush

```


If you want to think of points mainly as triples then you can convert a list of points into a sequence of monadic statements by first mapping every triple into a vertex, e.g. by:

```
map (\(x,y,z)->vertex$Vertex3 x y z)
```

and then combining the sequence of monadic statements into one monadic statement. Therefore you can use the standard function for monads: `sequence_`. The standard function `mapM_` is simply the composition of `map` and `sequence_`, such that a list of triples can be converted to a monadic vertex statement by:

```
mapM_ (\(x,y,z) -> vertex$Vertex3 x y z)
```

which is the technique used in the introductory example.

Beispiel:

Thus we can rewrite a points example in the following way: points are defined as a list of triples. Furthermore we define some useful auxilliary functions:

```

                                EvenMorePoints.hs
1  import Graphics.UI.GLUT
2  import Graphics.Rendering.OpenGL
3
4  main = do
5      getArgsAndInitialize
6      createAWindow "more points"
7      mainLoop
8
9  createAWindow windowName = do
10     createWindow windowName
11     displayCallback $= display
12
13  display = do
14     clear [ColorBuffer]
15     currentColor $= Color4 1 1 0 1
16     let points = [(0.1,0.6,0::GLfloat)
17                  ,(0.2,0.8,0)
18                  ,(0.3,0.1,0)
19                  ,(0,0,0)
20                  ,(0.4,-0.8,0)
21                  ,(-0.2,-0.8,0)
22                  ]
23     renderPoints points
24     flush
25
26  makeVertexes = mapM_ (\(x,y,z)->vertex$Vertex3 x y z)
27
28  renderPoints = renderAs Points
29
30  renderAs figure ps = renderPrimitive figure$makeVertexes ps

```

Some useful functions

In the following we want to explore all the other different shapes which can be rendered by OpenGL. All shapes are defined in terms of vertexes which you can think of as points.

We have already seen how to define vertexes and how to open a window and such things. We provide a simple module, which will be used in the consecutive examples. Some useful functions are defined in this module.

```

1 module PointsForRendering where
2   import Graphics.UI.GLUT
3   import Graphics.Rendering.OpenGL

```

A first function will open a window and use a given display function for the window graphics:

```

4 renderInWindow displayFunction = do
5   (progName,_) <- getArgsAndInitialize
6   createWindow progName
7   displayCallback $= displayFunction
8   mainLoop

```

The next function creates for a list of points, which are expressed as triples, and a basic shape a display function which renders the desired shape.

```

9 displayPoints points primitiveShape = do
10   renderAs primitiveShape points
11   flush
12
13 renderAs figure ps = renderPrimitive figure$makeVertexes ps
14
15 makeVertexes = mapM_ \(x,y,z)->vertex$Vertex3 x y z

```

Eventually we define a list of points as example and provide a function for easy use of these points:

```

16 mainFor primitiveShape
17   = renderInWindow (displayMyPoints primitiveShape)
18
19 displayMyPoints primitiveShape = do
20   clear [ColorBuffer]
21   currentColor $= Color4 1 1 0 1
22   displayPoints myPoints primitiveShape
23
24 myPoints
25   = [(0.2,-0.4,0::GLfloat)
26     ,(0.46,-0.26,0)
27     ,(0.6,0,0)
28     ,(0.6,0.2,0)
29     ,(0.46,0.46,0)
30     ,(0.2,0.6,0)
31     ,(0.0,0.6,0)

```

```

32     , (-0.26, 0.46, 0)
33     , (-0.4, 0.2, 0)
34     , (-0.4, 0, 0)
35     , (-0.26, -0.26, 0)
36     , (0, -0.4, 0)
37 ]

```

Beispiel:

We can now render the example points in a oneliner:

```

_____ RenderPoints.hs _____
1  import PointsForRendering
2  import Graphics.Rendering.OpenGL
3
4  main = mainFor Points

```

Lines

The next basic thing to do with vertexes is to connect them, i.e. consider them as starting and end point of a line. There are three ways to connect points with lines in OpenGL.

Singleton Lines The most natural way is to take pairs of points and draw lines between these. This is done in the primitive mode **Lines**. In order that this works properly an even number of vertexes needs to be supplied to the function **renderPrimitive**.

Beispiel:

Connecting our example points by lines. Pairs of points define singleton lines.

```

_____ RenderLines.hs _____
1  import PointsForRendering
2  import Graphics.Rendering.OpenGL
3
4  main = mainFor Lines

```

The resulting window can be found in figure 2.1.

Line Loops The next way to connect points with lines you probably can imagine is to make a closed figure. The end point of a line is the starting point of the next line and the last point is connected with the first, such that a closed loop of lines is created.

Beispiel:

Now we make a loop of lines with our example points.

```

_____ RenderLineLoop.hs _____
1  import PointsForRendering
2  import Graphics.Rendering.OpenGL
3
4  main = mainFor LineLoop

```

The resulting window can be found in figure 2.2.

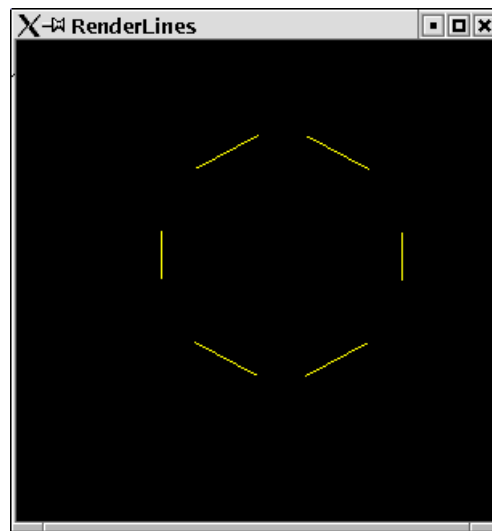


Figure 2.1: Lines between points.

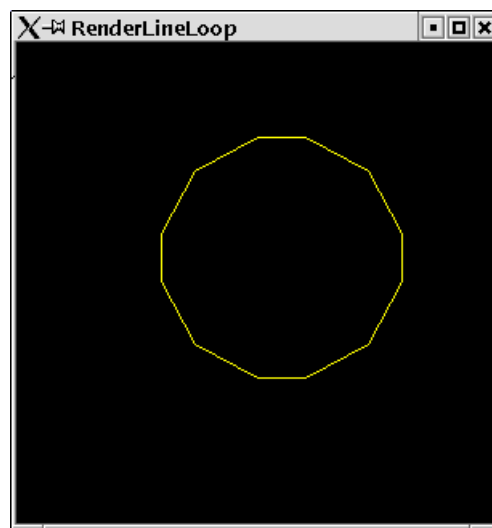


Figure 2.2: A loop of lines.

Line Strip A strip of lines is very close to a loop of lines. The only thing missing is the last line which connects the last point with the first one again.

Beispiel:

Now we make a strip of lines with our example points.

```
1 import PointsForRendering
2 import Graphics.Rendering.OpenGL
3
4 main = mainFor LineStrip
```

The resulting window can be found in figure 2.3.

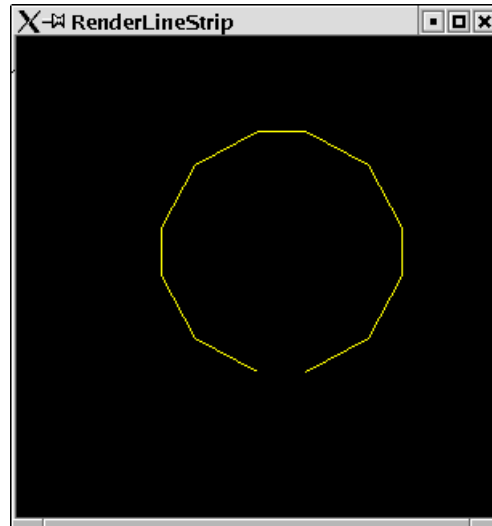


Figure 2.3: A strip in terms of lines.

Triangles

The next basic shape which can be rendered by OpenGL are triangles. Triples of points are taken and triangles are drawn with these. As for lines there are three flavours of triangles.

Triangle The most natural way of drawing triangles is to take triples and draw triangles. In order to work for triangles, the number of points provided needs to be a multiple of 3.

Beispiel:

Our example vertexes define 12 points such that we get 4 triangles

```
RenderTriangles.hs
1 import PointsForRendering
2 import Graphics.Rendering.OpenGL
3
4 main = mainFor Triangles
```

The resulting window can be found in figure 2.4.

Triangle Strips A triangle strip makes a sequence of triangles where the next triangle uses two points of its predecessor and one new point.

Beispiel:

For our 12 points a triangle strip will create 10 triangles.

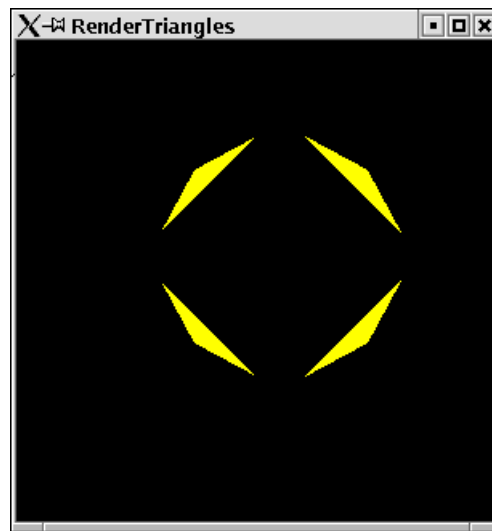


Figure 2.4: Triangles.

```
RenderTriangleStrip.hs
1 import PointsForRendering
2 import Graphics.Rendering.OpenGL
3
4 main = mainFor TriangleStrip
```

The resulting window can be found in figure 2.5.

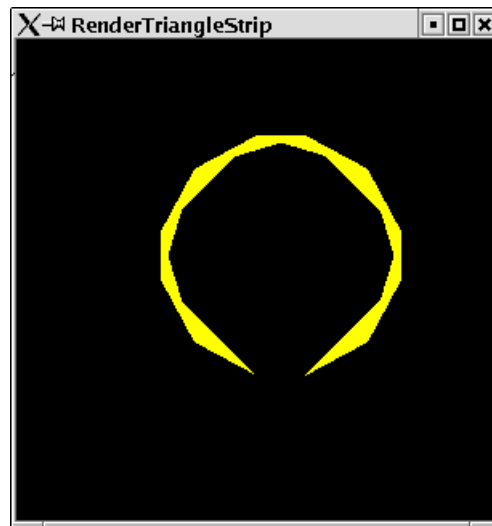


Figure 2.5: A triangle strip.

TriangleFan A fan has one starting point for all triangles. Triangles are always drawn starting from the first point.

Beispiel:

Our example points as a fan. 10 triangles are rendered.

```

1 import PointsForRendering
2 import Graphics.Rendering.OpenGL
3
4 main = mainFor TriangleFan

```

The resulting window can be found in figure 2.6.

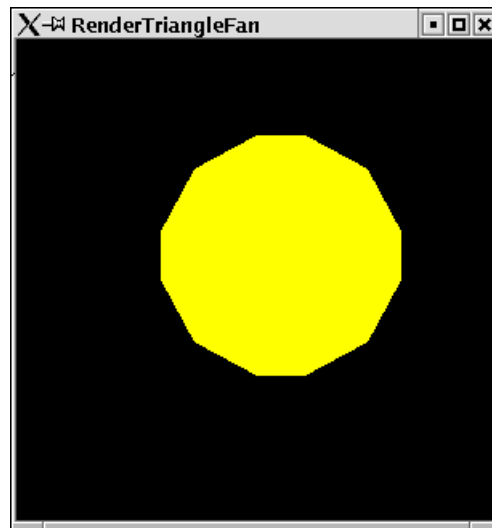


Figure 2.6: A triangle strip.

Quads

Lines connected two points, triangles three points, now we will connect four points. This is called a *quad*. There are two flavours of quads.

Singleton Quads The primitive mode **Quads** takes quadruples of points and connects them in order to render a filled figure.

Beispiel:

For our 12 example points OpenGL renders 3 quads

```

1 import PointsForRendering
2 import Graphics.Rendering.OpenGL
3
4 main = mainFor Quads

```

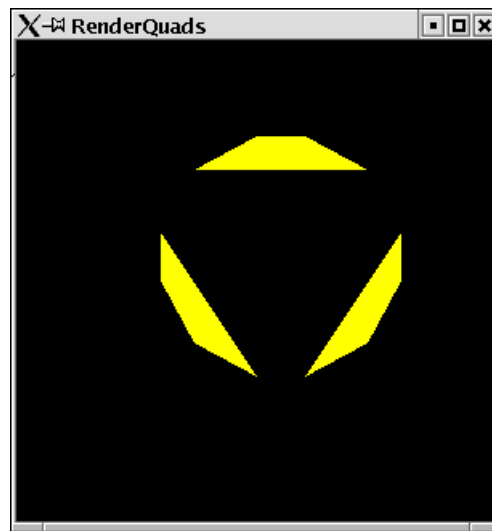


Figure 2.7: Quads.

The resulting window can be found in figure 2.7.

In a three dimensional world quads are unlike triangles not necessarily plane areas.

QuadStrips For a strip of quads OpenGL uses two points of the preceeding quads for the next quad. The number n of vertexes therefore needs to be of the form: $n = 4 + 2 * m$.

Beispiel:

Our examples vertexes now used for a strip of quads.

```

1  import PointsForRendering
2  import Graphics.Rendering.OpenGL
3
4  main = mainFor QuadStrip

```

The resulting window can be found in figure 2.8.³

Polygons

We connected two, three and four points. Eventually there is a shape that connects an arbitrary number of points. This is generally called a polygon. There are some restrictions for polygons:

- no convex corners are allowed.
- lines may not cross each other.

³Which somehow does not look like the expected?

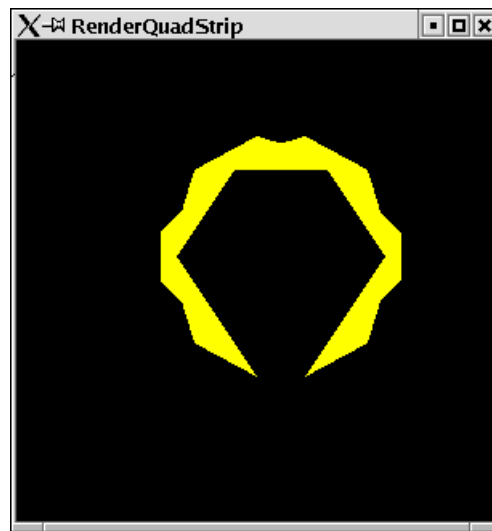


Figure 2.8: QuadStrips.

- polygons need to be planar.

Beispiel:

Eventually our vertexes are used to define a polygon.

```

1 import PointsForRendering
2 import Graphics.Rendering.OpenGL
3
4 main = mainFor Polygon

```

In this case the resulting window looks like the triangle fan we have seen before.

If you want to render polygons which hurt some of the restrictions above, you need to represent them by a set of smaller polygons. Since this is a tedious task to be done manually there is a library available, which does this for you: the *GLU* tessellation.

2.2.3 Curves, Circles and so on

In the last sections you have seen all primitive shapes, which can be rendered by OpenGL. Everything else needs to be constructed in term of these primitives. Especially you might wonder where curves and circles are. The bad news is: you have to do these by yourself.

Circles

With a bit mathematics you probably have already guessed how to do curves and especially circles. You need to approximate them with a large number of lines. If the lines get very small we eventually see a curve. Let us try this with circles. We write a module which gives us some utility functions for rendering circles.

```

1 module Circle where
2   import PointsForRendering
3   import Graphics.Rendering.OpenGL

```

The crucial function calculates a list of points which are all on the circle. You need a bit of basic geometrical knowledge for this. The coordinates of the points on a circle can be determined by $\sin(\alpha)$ and $\cos(\alpha)$ where α is between 0 and 2π .

Thus we can easily calculate the coordinates of an arbitrary number of points on a circle:

```

4 circlePoints radius number
5   = [let alpha = twoPi * i / number
6       in (radius*(sin (alpha)) ,radius * (cos (alpha)),0)
7       | i <- [1,2..number]]
8   where
9       twoPi = 2*pi

```

If we take a large enough number then we will eventually get a circle:

```

10 circle radius = circlePoints radius 100

```

The following function can be used to render the circle figures:

```

11 renderCircleApprox r n
12   = displayPoints (circlePoints r n) LineLoop
13
14 renderCircle r = displayPoints (circle r) LineLoop
15 fillCircle r = displayPoints (circle r) Polygon

```

Beispiel:

First we test what kind of shape we get for small approximation numbers.

```

1 import PointsForRendering
2 import Circle
3
4 import Graphics.Rendering.OpenGL
5
6 main = renderInWindow $ do
7     clear [ColorBuffer]
8     renderCircleApprox 0.8 10

```

The resulting graphic can be seen in figure 2.9.

Beispiel:

Now we can test, if the resulting circle is, what we expected.

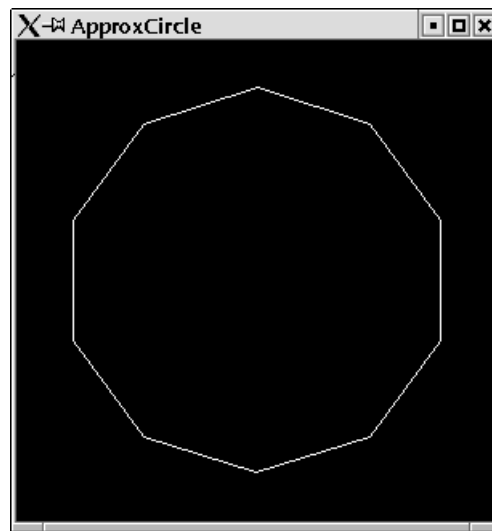


Figure 2.9: 10 points on a circle.

```
TestCircle.hs
1 import PointsForRendering
2 import Circle
3
4 import Graphics.Rendering.OpenGL
5
6 main = renderInWindow $ do
7     clear [ColorBuffer]
8     renderCircle 0.8
```

The resulting graphic can be seen in figure 2.10.

Beispiel:

And eventually have a look at the filled circle.

```
FillCircle.hs
1 import PointsForRendering
2 import Circle
3
4 import Graphics.Rendering.OpenGL
5
6 main
7   = renderInWindow $ do
8     clear [ColorBuffer]
9     fillCircle 0.8
```

The resulting graphic can be seen in figure 2.11.

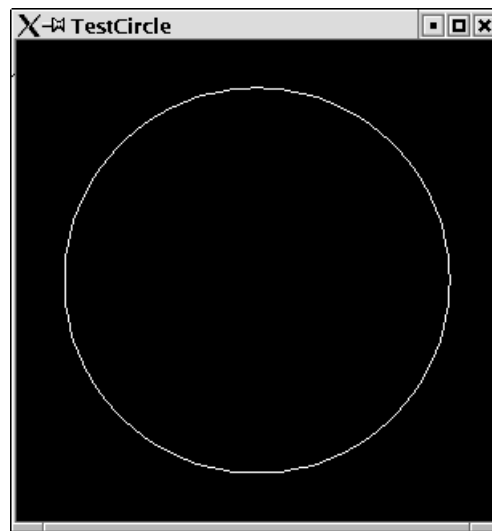


Figure 2.10: Rendering a full circle.

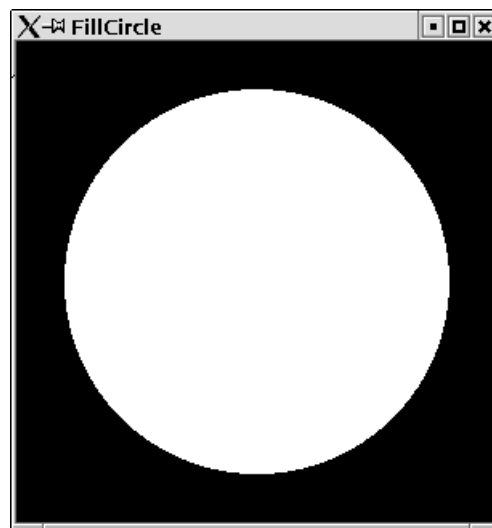


Figure 2.11: A filled circle.

Rings

Now, where you know how to do circles, you can equally as easy define functions for rendering rings. A ring has an inner and an outer circle and fills the space between these. So we can approximate these two rings and render quads between them.

```
Ring.hs
1 module Ring where
2
3 import PointsForRendering
4 import Circle
```

```
5 | import Graphics.Rendering.OpenGL
```

We can simply define the points of the inner and outer ring and merge these. The resulting list of points can then be rendered as a `QuadStrip`. Since there is no primitive mode for quad loops, we need to append the first two points as the last points again:

```
6 | ringPoints innerRadius outerRadius
7 | = concat$map (\(x,y)->[x,y]) (points++[p])
8 |   where
9 |     innerPoints = circle innerRadius
10 |    outerPoints = circle outerRadius
11 |    points@(p:_) = zip innerPoints outerPoints
```

Eventually we provide a small function for rendering ring shapes.

```
12 | ring innerRadius outerRadius
13 | = displayPoints (ringPoints innerRadius outerRadius) QuadStrip
```

Beispiel:

We can test the ring functions:

```
1 | import PointsForRendering
2 | import Ring
3 |
4 | import Graphics.Rendering.OpenGL
5 |
6 | main = renderInWindow $ do
7 |     clear [ColorBuffer]
8 |     ring 0.7 0.9
```

The resulting graphic can be seen in figure 2.12.

2.2.4 Attributes of primitives

There are some more attributes that can be set for primitive shapes (besides the color, which we have already set).

Point Size

You could argue that there is no need for single points. A point can be modelled by a circle that has a small radius (or in the third dimension a sphere). However, there is something like a point in OpenGL and you can set its size. This size value for points does not refer to a radius in the coordinate system but is measured in terms of screen pixels. The default value is, one pixel per point.

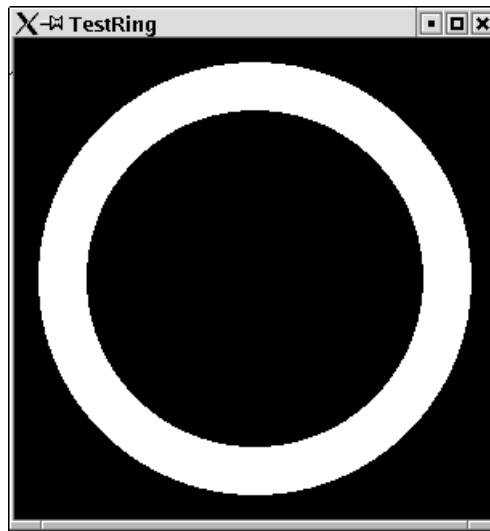


Figure 2.12: A simple ring shape.

Beispiel:

We set the point size to 10 pixels:

```

1  import Graphics.Rendering.OpenGL
2  import PointsForRendering
3
4  main = renderInWindow display
5
6  display = do
7      pointSize $= 10
8      displayMyPoints Points

```

The resulting graphic can be seen in figure 2.13.

Line Attributes

As for points, there are also further attributes for lines. First of all there is a line width. As for the point size, this is measured in screen pixels. Furthermore, you can set some line stipple: this is the pattern of the line, dashes etc. For the line stipple there is a state variable of type: `Maybe (GLint, GLushort)`. The second argument of the value pair denotes the kind of stipple. For every short value there is one stipple. The short value has 16 bits. Every bit stands for a pixel. If for the corresponding short number the bit is set, then the pixel will be drawn, otherwise not. This means that for the short number 0 you will not see anything of your line, and for the value 65535 you will see a solid line.

The integer number of the value pair denotes a factor for the chosen stipple. For some positiv integer n every bit of the short number stands for n bits.

Beispiel:

Setting the width of lines and a stipple:

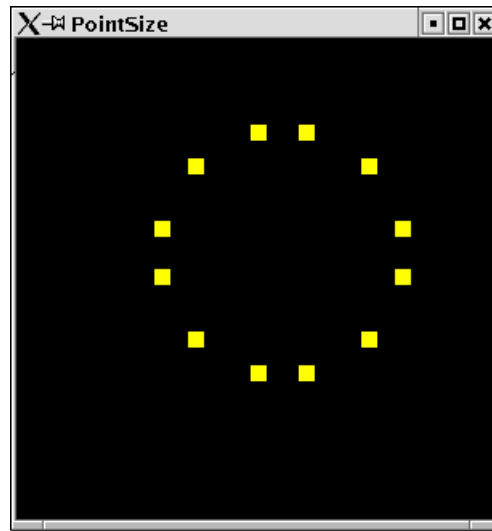


Figure 2.13: Points of a large size.

```

----- LineAttributes.hs -----
1  import Graphics.Rendering.OpenGL
2  import Graphics.UI.GLUT as GLUT
3  import PointsForRendering
4
5  main = renderInWindow display
6
7  display = do
8      clearColor $= Color4 1 1 1 1
9      clear [ColorBuffer]
10     lineStipple $= Just (1,255)
11     currentColor $= Color4 0 0 0 1
12     lineWidth $= 10
13     displayPoints squarePoints LineLoop
14     flush
15
16     squarePoints
17     = [(-0.7,-0.7,0),(0.7,-0.7,0),(0.7,0.7,0),(-0.7,0.7,0)]

```

The resulting graphic can be seen in figure 2.14.

Colors

You might have wondered, why the function `renderPrimitive` takes monadic statements as argument and not simply a list of vertexes? This means we could pass any monadic statement to the function `renderPrimitive`, not only statements that define vertexes by the call of the function `vertex`. There are some statements, which are allowed in the statements passed to `renderPrimitive`. One of these is setting the current color before every call of `vertex`

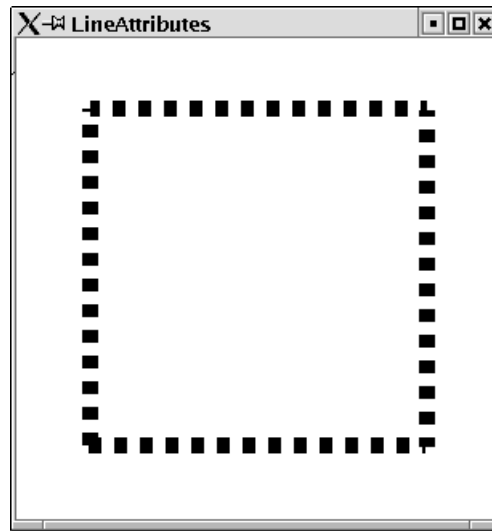


Figure 2.14: Thick stippled lines.

to a new value. When finally rendering the primitive, OpenGL takes these color values into account.

Beispiel:

We define a triangle. Before the three vertexes of the triangle are defined, the current color is set to a new value.

```

1  import Graphics.Rendering.OpenGL
2  import Graphics.UI.GLUT as GLUT
3  import PointsForRendering
4
5  colorTriangle = do
6    setCurrentColor $= Color4 1 0 0 1
7    vertex$Vertex3 (-0.5) (-0.5) (0::GLfloat)
8    setCurrentColor $= Color4 0 1 0 1
9    vertex$Vertex3 (0.5) (-0.5) (0::GLfloat)
10   setCurrentColor $= Color4 0 0 1 1
11   vertex$Vertex3 (-0.5) (0.5) (0::GLfloat)
12
13  main = renderInWindow display
14
15  display = do
16    clearColor $= Color4 1 1 1 1
17    clear [ColorBuffer]
18    renderPrimitive Triangles colorTriangle
19    flush

```

The resulting window can be found in figure 2.15.

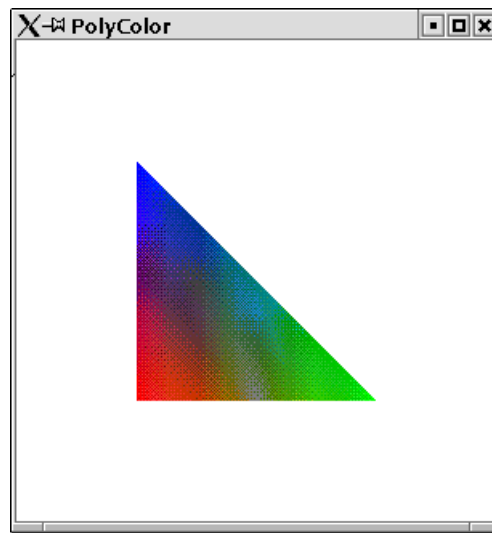


Figure 2.15: A triangle with different vertex colors

2.2.5 Tessellation

Rendering of polygons is very limited. We cannot render polygons for crossing lines, or convex corners. Such polygons need to be expressed by a set of simpler polygons. In the module `Graphics.Rendering.OpenGL.GLU.Tessellation` there are a number of functions, which calculate a set of simpler polygons. For the time being, we will not go into detail, but give one single example, of how to use this library.

Beispiel:

We want to render stars. These are shapes with convex corners.

```

1 module Star where
2   import Graphics.Rendering.OpenGL
3   import Graphics.UI.GLUT as GLUT
4   import Data.Either
5   import Circle
6   import List

```

We can easily calculate the points on the star rays. They are all on one circle. We can use our function for defining circle points and get a list of points. For rendering the star, we take first the points with odd index followed by the points with even index.

```

7   starPoints radius rays
8   = map (\(_, (x,y,z)) -> Vertex3 x y z) (os++es)
9   where
10      (os,es) = partition (\(i,_)-> odd i)
11                  $zip [1,2..]
12                  $circlePoints radius rays

```

For tessellation we need to create a `ComplexPolygon`, which has a list of `ComplexContour`. A `ComplexContour` contains a list of `AnnotatedVertexes`. The annotation can be used for color or similar information. We do not make use of this annotation and simply annotate every vertex with 0.

```

13 complexPolygon points
14   = ComplexPolygon
15     [ComplexContour $map (\v->AnnotatedVertex v 0) points]

```

The function `tessellate` creates a list of simple polygons. It needs some control information, which we do not explain here.

```

16 star radius rays= do
17   (Right startess)
18   <- tessellate
19       TessWindingPositive 0 (Normal3 0 0 0) noOpCombiner
20       $complexPolygon (starPoints radius rays)
21   drawSimplePolygon startess

```

The resulting simple polygons can be rendered with the function `renderPrimitive`.

```

22 drawSimplePolygon (SimplePolygon primitiveParts) =
23   mapM_ renderPrimitiveParts primitiveParts
24
25 renderPrimitiveParts (Primitive primitiveMode vertices) =
26   renderPrimitive primitiveMode
27   $mapM_ (vertex . stripAnnotation) vertices
28
29 stripAnnotation (AnnotatedVertex plainVertex _) = plainVertex
30
31 noOpCombiner _newVertex _weightedProperties = 0.0 ::GLfloat

```

Now we can test our stars. We render two stars, one with 7 and one with 5 rays.

```

1 import PointsForRendering
2 import Graphics.Rendering.OpenGL
3 import Graphics.UI.GLUT as GLUT
4 import Star
5
6 main = renderInWindow$do
7   clearColor $= Color4 1 1 1 1
8   clear [ColorBuffer]
9
10  currentColor $= Color4 1 0 0 1
11  star 0.9 7
12
13  currentColor $= Color4 1 1 0 1
14  star 0.4 5

```

The resulting window can be found in figure 2.16.

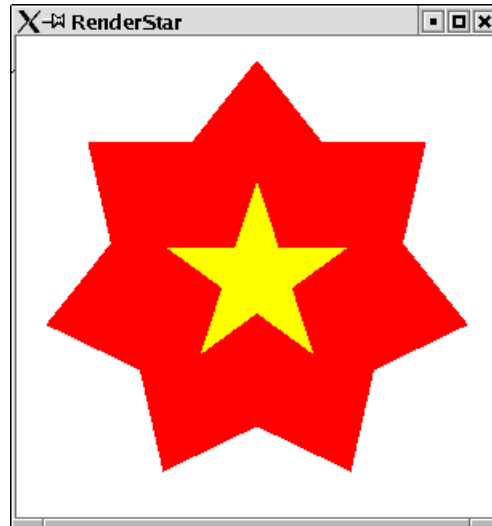


Figure 2.16: RenderStar

2.2.6 Cubes, Dodecahedrons and Teapots

The bad news was that just very basic shapes are provided by OpenGL for rendering. The good news is that the OpenGL library comes along with a library that contains a large number of shapes.

Beispiel:

You probably need very often the shape of a teapot. Since this is so elementary a library function is provided for this.

```
Tea.hs
1 import Graphics.UI.GLUT
2 import Graphics.Rendering.OpenGL
3
4 import PointsForRendering
5
6 main = renderInWindow display
7
8 display = do
9     clear [ColorBuffer]
10    renderObject Solid$ Teapot 0.6
11    flush
```

The resulting graphic can be seen in figure 2.17.



Figure 2.17: A tea pot.

Chapter 3

Modelling Transformations

By now you know, how to define different shapes for rendering. You might wonder how to place shapes on special positions or how to scale or rotate your shapes. This is done by so called transformation matrixes. Before something is rendered by OpenGL a transformation operation is performed on it. Every point will get multiplied with the transformation matrix. The transformation matrix is part of the state. So in order to transform a shape in some way, first the transformation matrix has to be set and then the shapes are to be rendered. If not specified otherwise the transformation matrix is the identity operation, i.e. no transformation is performed. You can always reset the transformation matrix to the identity by the call of the monadic statement `loadIdentity`. Then the current matrix is discarded and no transformation is applied to the next rendering operations.

3.1 Translate

One transformation is to move a shape to another position. The according matrix is set by the statement `translate`. It has one argument: a vector of size three which denotes in which direction the following shapes are to be moved. Every vertex that will be rendered after a `translate` statement will be moved by the values of this vector.

Beispiel:

The function `ring` we defined before only defined rings which have the center coordinates $(0,0,0)$. If we want to place rings somewhere else then we need to apply a translate matrix.

```
----- SomeRings.hs -----  
12 import PointsForRendering  
13 import Ring  
14 import Graphics.Rendering.OpenGL
```

We define a function, which creates a ring at a given position. Therefore we first set the transformation to the translate transformation then define the ring and finally set the transformation matrix back to the identity:

```

15 ringAt x y innerRadius outerRadius = do
16     translate$Vector3 x y (0::GLfloat)
17     ring innerRadius outerRadius

```

We can test this by placing some ring in different colors on the screen.

```

18 main = do
19     renderInWindow someRings
20
21 someRings = do
22     clearColor $= Color4 1 1 1 1
23     clear [ColorBuffer]
24
25     loadIdentity
26     currentColor $= Color4 1 0 0 1
27     ringAt 0.5 0.3 0.1 0.12
28
29     loadIdentity
30     currentColor $= Color4 0 1 0 1
31     ringAt (-0.5) 0.3 0.3 0.5
32
33     loadIdentity
34     currentColor $= Color4 0 0 1 1
35     ringAt (-1) (-1) 0.7 0.75
36
37     loadIdentity
38     currentColor $= Color4 0 1 1 1
39     ringAt 0.7 0.7 0.2 0.3

```

The resulting graphic can be seen in figure 3.1.

Note that if we did not reset the transformation back to the identity, we would get the composition of all transformations.

3.2 Rotate

Another transformation that can be performed is rotation. The rotate statement has two arguments. The first one specifies by which degree the following shapes are to be rotated counterclockwise. The second argument is a vector which specifies around which axis the shape is to be rotated.

Beispiel:

In this example we apply the composition of two transformations. Squares are moved to some position and furthermore rotated around the z -axis.

We write a simple module for rendering filled rectangles:

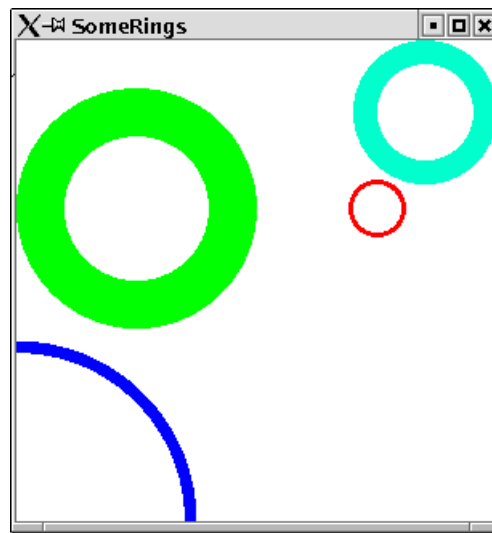


Figure 3.1: Rings translated to different positions.

```

1 module Squares where
2
3 import Graphics.Rendering.OpenGL
4 import PointsForRendering

```

Here is a function for arbitrary rectangles:

```

5
6 myRect width height =
7   displayPoints [(w,h,0),(w,-h,0),(-w,-h,0),(-w,h,0)] Quads
8   where
9     w = width/2
10    h = height/2

```

A square is just a special case:

```

11 square width = myRect width width

```

Now we will transform squares.

```

1 import PointsForRendering
2 import Squares
3 import Graphics.Rendering.OpenGL

```

We define a function, which applies the rotate transformation to a square. It is rotated around the z -axis.

```

1
2
3
4 rotatedSquare alpha width = do
5   rotate alpha $Vector3 0 0 (1::GLfloat)
6   square width

```

A further utility function moves some shape to a specified position. Note that this function resets the matrix again.

```

7
8
9
10 displayAt x y displayMe = do
11   translate$Vector3 x y (0::GLfloat)
12   displayMe
13   loadIdentity

```

Some squares are defined and rotated:

```

11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
SomeSquares.hs
main = do
  renderInWindow someSquares

someSquares = do
  clearColor $= Color4 1 1 1 1
  clear [ColorBuffer]

  currentColor $= Color4 1 0 0 1
  displayAt 0.5 0.3$rotatedSquare 15 0.12

  currentColor $= Color4 0 1 0 1
  displayAt (-0.5) 0.3$rotatedSquare 25 0.5

  currentColor $= Color4 0 0 1 1
  displayAt (-1) (-1)$rotatedSquare 4 0.75

  currentColor $= Color4 0 1 1 1
  displayAt 0.7 0.7$rotatedSquare 40 0.3

```

The resulting graphic can be seen in figure 3.2.

3.3 Scaling

The third transformation enables you to scale shapes. This is not only useful for changing the size of some object but for stretching it in some direction. The transformation `scale` has three arguments, which represent the scaling factors in the three dimensional space.

Beispiel:

We apply three transformations on the tea pot example. We rotate and translate it and finally we stretch it a bit by a scale transformation.

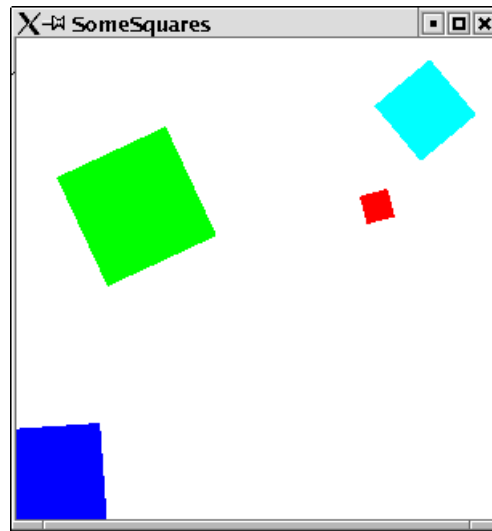


Figure 3.2: Squares translated and rotated.

```

1  import Graphics.UI.GLUT
2  import Graphics.Rendering.OpenGL
3
4  import PointsForRendering
5  main = renderInWindow display
6
7  display = do
8      clear [ColorBuffer]
9      scale 0.3 0.9 (0.3::GLfloat)
10     translate$Vector3 (-0.3) 0.3 (0::GLfloat)
11     rotate 30 $Vector3 0 1 (0::GLfloat)
12     renderObject Solid$ Teapot 0.6
13     loadIdentity
14     flush

```

The resulting graphic can be seen in figure 3.3. As you see it looks now like a coffee pot.

Remember that the scale and the rotate transformation always refer to the origin $(0,0,0)$ of your coordinates. Rotating an object, which is not situated at the origin will move it around the origin. Scaling an object which is not situated at the origin might deform the object in surprising ways.

3.4 Composition of Transformations

Since Haskell is a functional programming language let us think of transformations as functions. A transformation is a function that is applied to every vertex before it is rendered.

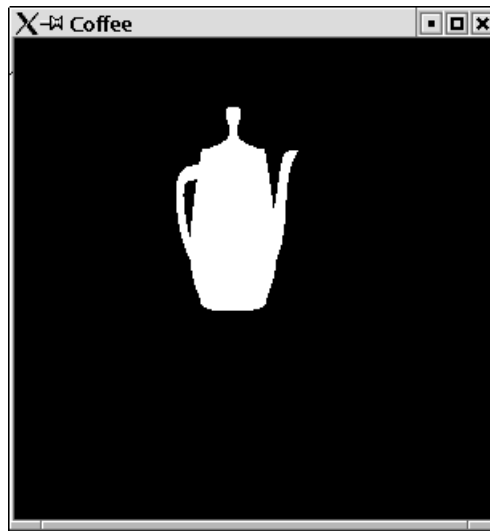


Figure 3.3: A coffee pot.

If you define two transformations for an object, e.g. a rotation and a translation, then you define a composition of these transformations.

The code:

```

1  rotatedSquareAt width alpha x y z = do
2    translate$Vector3 x w y
3    rotate alpha $Vector3 0 0 (1::GLfloat)
4    square width

```

defines a composition of a translate und a rotate transformation, which is applied to a square figure. A sequence of transformation statements is composed to a single transformation in the same way as the standard function composition operator $(.)$ composes functions: $(f . g) x = f(g(x))$. The compositional function $(f . g)$ is the same as first applying function g and then applying f . For transformations in HOpenGL this means that for a sequence of transformations

```

1  translate$Vector3 x w y
2  rotate alpha $Vector3 0 0 (1::GLfloat)

```

first the points are rotated and then they are translated.

The order in which transformations are performed is of course not arbitrary. A rotation after a translation is different to a translation after a rotation.

Beispiel:

This example illustrates the different compositions of rotation and translation.

```

_____ Compose.hs _____
1  import PointsForRendering
2  import Squares

```

```
3 import Graphics.Rendering.OpenGL
4
5 displayAt x y displayMe = do
6     displayMe
7     loadIdentity
8
9 main = do
10     renderInWindow someSquares
11
12 someSquares = do
13     clearColor $= Color4 1 1 1 1
14     clear [ColorBuffer]
```

A black square at the origin:

```
15 Compose.hs
16     clearColor $= Color4 0 0 0 1
17     square 0.5
18     loadIdentity
```

A blue square translated:

```
19 Compose.hs
20     clearColor $= Color4 0 0 1 1
21     translate$Vector3 0.5 0.5 (0::GLfloat)
22     square 0.5
23     loadIdentity
```

A light blue square that is rotated:

```
24 Compose.hs
25     clearColor $= Color4 0 1 1 1
26     rotate 35 $Vector3 0 0 (1::GLfloat)
27     square 0.5
28     loadIdentity
```

A red square that is first rotated and then translated:

```
29 Compose.hs
30     clearColor $= Color4 1 0 0 1
31     translate$Vector3 0.5 0.5 (0::GLfloat)
32     rotate 35 $Vector3 0 0 (1::GLfloat)
33     square 0.5
34     loadIdentity
```

A yellow square that is first translated and then rotated:

```
35 Compose.hs
36     clearColor $= Color4 1 1 0 1
37     rotate 35 $Vector3 0 0 (1::GLfloat)
38     translate$Vector3 0.5 0.5 (0::GLfloat)
39     square 0.5
40     loadIdentity
```

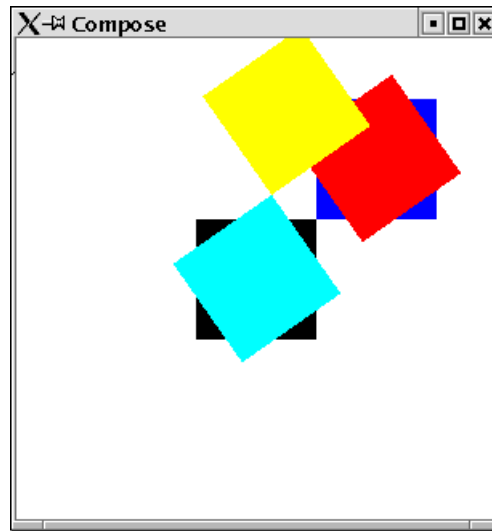


Figure 3.4: Different compositions of translation and rotation.

The resulting window can be found in figure 3.4.

Since the scale and the rotate transformation refer both to the origin and the translate transformation can move objects away from the origin it is a good policy to create objects at the origin, then rotate and scale it and finally translate it to its final position. Therefore predefined shapes in the library are usually positioned at the origin, as e.g. the tea pot.

3.5 Defining your own transformation

The three ready to use transformations rotation, scaling and translation or their composition might not suffice for your needs. Then you can define your own transformations. Technically a transformation in OpenGL is represented as a matrix. Every vertex gets multiplied by the transformation matrix before it is rendered. In order to define a transformation, we will need to construct such a matrix.

Internally every vertex in OpenGL is not represented by 3 coordinates (x, y, z) but by four coordinates (x, y, z, w) . The x, y, z values are divided by w . Usually the value of w is 1.0.

Thus for a transformation matrix you need a matrix of four rows and four columns. Remember that a matrix is multiplied with a vector in the following way:

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x_{11} * x + x_{12} * y + x_{13} * z + x_{14} * w \\ x_{21} * x + x_{22} * y + x_{23} * z + x_{24} * w \\ x_{31} * x + x_{32} * y + x_{33} * z + x_{34} * w \\ x_{41} * x + x_{42} * y + x_{43} * z + x_{44} * w \end{pmatrix}$$

OpenGL provides a function for creation of a transformation matrix out of a list: `matrix`. It takes as first argument a parameter, which specifies in which order the matrix elements appear in the list: `RowMajor` for row wise and `ColumnMajor` for column wise appearance. The function `multMatrix` allows to multiply your newly created transformation matrix to the current transformation context.

3.5.1 Shear

We can now define our own transformations. We can define the transformation *shear*. Mathematical textbooks define *shear* in the following way:

A transformation in which all points along a given line L remain fixed while other points are shifted parallel to L by a distance proportional to their perpendicular distance from L. Shearing a plane figure does not change its area.

Eric Weissteins's world of mathematics (<http://mathworld.wolfram.com/Shear.html>)

We define a *shear* transformation, which leaves *y* and *z* coordinates unchanged, and adds to the *x* coordinate some value depending on the value of *y*. For some *f* we need the following transformation matrix:

$$\begin{pmatrix} 1 & f & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x + f * y \\ y \\ z \\ w \end{pmatrix}$$

As you can see, this is almost the identity. We can define this in HOpenGL:

```

----- MyTransformations.hs -----
1 module MyTransformations where
2 import Graphics.Rendering.OpenGL
3 import Graphics.UI.GLUT as GLUT
4
5 shear f =
6   let m :: GLmatrix GLfloat
7       m = (matrix RowMajor [1,f,0,0
8                             ,0,1,0,0
9                             ,0,0,1,0
10                            ,0,0,0,1])
11   in multMatrix m

```

Let us test our new transformation:

```

----- TestShear.hs -----
1 import PointsForRendering
2 import Circle
3 import Squares
4 import MyTransformations
5
6 import Graphics.Rendering.OpenGL
7 import Graphics.UI.GLUT as GLUT
8
9 main = renderInWindow$do
10   loadIdentity
11   clearColor $= Color4 1 1 1 1
12   clear [ColorBuffer]
13   translate$Vector3 0.5 0.5 (0::GLfloat)
14   shear 0.5

```

```

15  currentColor $= Color4 0 0 1 1
16  fillCircle 0.5
17
18  loadIdentity
19  translate$Vector3 (-0.5) (-0.5) (0::GLfloat)
20  shear 0.5
21  currentColor $= Color4 1 0 0 1
22  square 0.5

```

The resulting window can be found in figure 3.5.

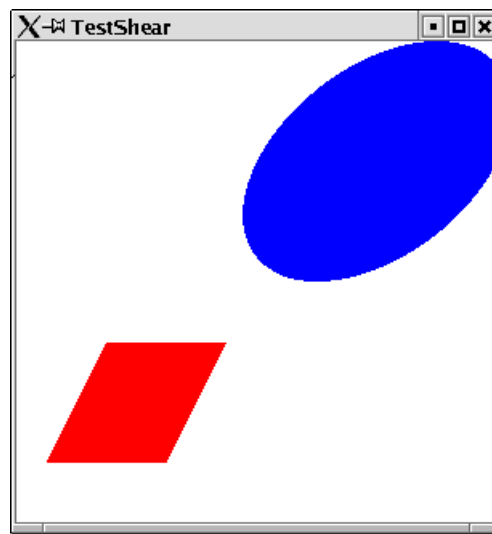


Figure 3.5: Applying shear to some shapes.

3.6 Some Word of Warning

You might get strange effects when you forget to reset the transformation matrix. This might not only effect further rendering statements but also applies to the redisplay of your window. The display function you specified for your window will be called whenever the window needs to be displayed. However this does not automatically reset the transformation matrix to the identity matrix. This results in the effect that every redisplay of your window changes its contents.

Beispiel:

In this example a ring is displayed. Each time the display function is called the contents of the ring moves a bit. Compile the program and hide the resulting window behind some other window. You will observe how the ring moves within the window, until it is no longer displayed.

```

_____ ForgottenReset.hs _____
1  import Graphics.UI.GLUT
2  import Graphics.Rendering.OpenGL

```

```

3
4 import PointsForRendering
5 import Ring
6 import Squares
7
8 main = renderInWindow display
9
10 display = do
11   clear [ColorBuffer]
12   translate$Vector3 (-0.1) 0.1 (0::GLfloat)
13   ring 0.2 0.4
14   flush

```

As a matter of fact this effect may not only occur with transformations, but every state changing statement. If you set the color as last statement in your display function to some value then this will be the current color in the next call of the display function. Thus it is better to ensure that the display function leaves a clean state, i.e. the state it expects to find, when it is called, or even better let the display functions not rely on any previously set states.

3.7 Local transformations

Often you will have the situation, that you are in a context of some transformations. Maybe for certain parts of you shape you want to add some further transformation but for other parts return to the outer transformation context. In such situations you cannot use the statement `loadIdentity` since this will not only delete the transformations you wanted to be applied to your local part of the the complete shape but the whole transformation context.

HOpenGL provides a function which allows to add some more transformations to some local parts of your shape. This function is called `matrixExcursions` which refers to the fact that transformations are technically implemented as matrixes. `matrixExcursion` has one argument, which is a monadic statement. The application of `matrixExcursion` is a monadic statement:

```

1 matrixExcursion :: IO a -> IO a

```

Every transformation done within this monadic statement will not be done only locally. It does not effect the statements which follow after the application of `matrixExcursion`.

Beispiel:

To demonstrate the use of `matrixExcursion` we provide a module, which is able to render a side of the famous Rubik's Cube. Such a side consists of 9 squares which are of some color and which have a black frame. We can render such a shape, by rendering the single framed squares at the origin and then move them to their position. This movement is done within a `matrixExcursion` application.

```

1 module RubikFace where
2   import Graphics.UI.GLUT
3   import Graphics.Rendering.OpenGL
4
5   import Squares
6   import PointsForRendering

```

Doing a frame involves the four sides of a frame. Each side is created at the origin and then moved to its final position:

```

7   frame width height border = do
8     let bh = border/2
9     let wh = width/2-bh
10    let hh = height/2-bh
11
12    matrixExcursion $ do
13      translate $Vector3 0 hh (0::GLfloat)
14      myRect width border
15    matrixExcursion $ do
16      translate $Vector3 0 (-hh) (0::GLfloat)
17      myRect width border
18    matrixExcursion $ do
19      translate $Vector3 (-wh) 0 (0::GLfloat)
20      myRect border height
21    matrixExcursion $ do
22      translate $Vector3 wh 0 (0::GLfloat)
23      myRect border height

```

Each of the nine fields is rendered by drawing its frame and its colored square:

```

24   originField width color = do
25     let frameWidth = width/10
26     currentColor $= Color4 0 0 0 1
27     frame width width frameWidth
28     let sc = 18/20::GLfloat
29     currentColor $= color
30     square (width-frameWidth)

```

Eventually the side of Rubik's Cube can be drawn

```

31   renderArea :: GLfloat -> [[Color4 GLfloat]] -> IO ()
32   renderArea width css
33   = do
34     let cs = concat css
35     cps = zip cs $ areaFields width
36     mapM_ (\(c,f)-> f(originField width c)) cps
37
38   areaFields width =

```



```

39 [makeSquare x y |x<-[-1,0,-1],y<-[-1,0,-1]]
40   where
41     makeSquare xn yn = \f -> matrixExcursion $ do
42       let
43         x = xn*width
44         y = yn*width
45         translate $Vector3 x y 0
46         f
47
48 red    = Color4 1 0 0 (1::GLfloat)
49 green  = Color4 0 1 0 (1::GLfloat)
50 blue   = Color4 0 0 1 (1::GLfloat)
51 yellow = Color4 1 1 0 (1::GLfloat)
52 white  = Color4 1 1 1 (1::GLfloat)
53 black  = Color4 0 0 0 (1::GLfloat)

```

The following module tests the rendering. Two sides are rendered. Further transformations are applied to them.

```

_____ RenderRubikFace.hs _____
1  import PointsForRendering
2  import Graphics.Rendering.OpenGL
3
4  import PointsForRendering
5  import RubikFace
6
7  _FIELD_WIDTH :: GLfloat
8  _FIELD_WIDTH = 1/5
9
10 main = renderInWindow faces
11
12 faces = do
13   clearColor $= white
14   clear [ColorBuffer]
15
16   loadIdentity
17   translate $Vector3 (-0.6) 0.4 (0::GLfloat)
18   renderArea _FIELD_WIDTH r1
19
20   loadIdentity
21   translate $Vector3 (0.1) (-0.3) (0::GLfloat)
22   rotate 290 $ Vector3 0 0 (1::GLfloat)
23   scale 1.5 1.5 (1::GLfloat)
24   renderArea _FIELD_WIDTH r1
25
26 r1=[[red,blue,yellow],[white,green,red],[green,yellow,blue]]

```

The resulting window can be found in figure 3.6.

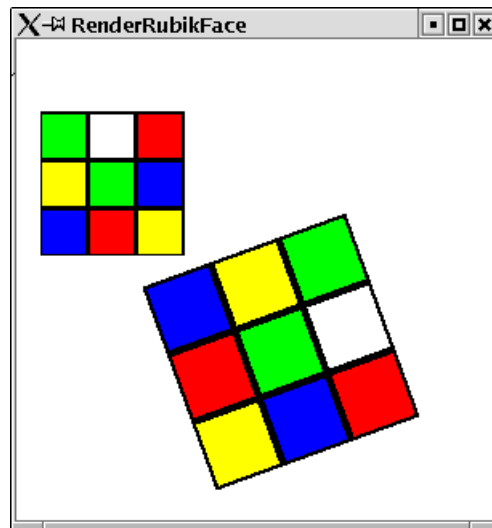


Figure 3.6: a Side of Rubik's Cube with further transformations applied to it.

Chapter 4

Projection

4.1 The Function Reshape

Up to now we always relied on the default values for most attributes which are concerned with projection. From where do we look at the scenery? Which coordinates are displayed to what extend on the screen. Such attributes can be set in the reshape callback function. This function gets the window size as argument and specifies which coordinates are to be seen on the screen. At first glance the name seems to be a bit misleading, since it evokes the image that it is just called, when someone resizes the window. The first time the reshape function is called is at the opening of the window.

The reshape function might be empty. This is modelled by the Haskell data type `Maybe`.

Beispiel:

We define the first reshape function for a window. It is the identity function, which does not specify anything, how to render the picture.

```
Reshape1.hs
1 import Graphics.UI.GLUT
2 import Graphics.Rendering.OpenGL
3
4 import PointsForRendering
5
6 main = do
7     (progName,_) <- getArgsAndInitialize
8     createWindow progName
9     displayCallback $= display
10    reshapeCallback $= Just reshape
11    mainLoop
12
13 display = do
14     clear [ColorBuffer]
15     displayPoints points Quads
16     where
17         points
18             = [(0.5,0.5,0)
```

```

19         , (-0.5, 0.5, 0)
20         , (-0.5, -0.5, 0)
21         , (0.5, -0.5, 0)]
22
23 reshape s = return ()

```

Run this example. You will see a white square in the middle of a black screen. Now resize the window. You will notice that the size of the square will not change. If you make the window smaller parts of the picture are not displayed, if you enlarge the window parts of the window contain no image (which means it might be some arbitrary image). Figure 4.1 shows how the window looks after enlarging it a bit.

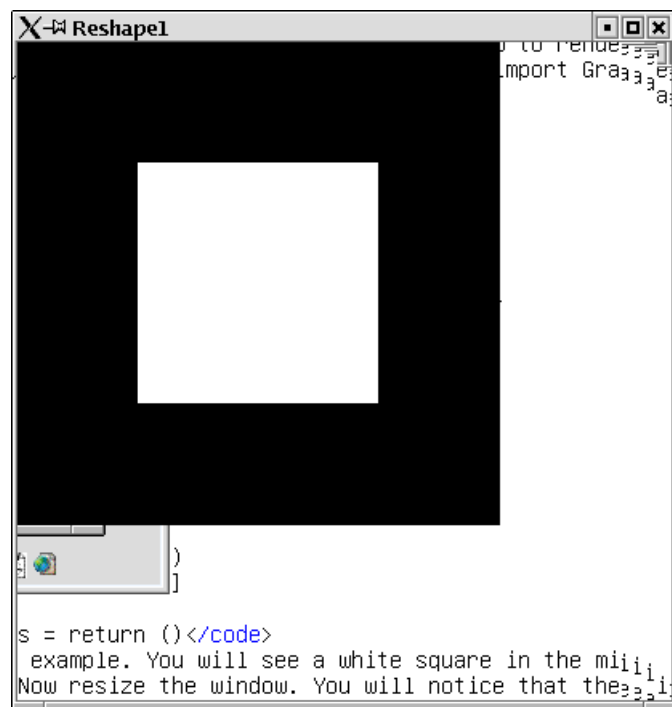


Figure 4.1: Enlarging a window with the empty reshape function.

4.2 Viewport: The Visible Part of Screen

Usually you want to define in the reshape function, which parts of the window pane are to be used for rendering the picture. There is a state variable **viewport**, which contains exactly this information. It is a pair, of a position and a size. The position is the offset from the upper left corner in pixels. The size is the size of the screen to be used for rendering in pixels.

Beispiel:

If you want the window to be used completely for rendering the image, then the

position needs to be set to `Position 0 0`. i.e. no offset and as size the complete window size is to be used:

```

1  import Graphics.UI.GLUT
2  import Graphics.Rendering.OpenGL
3
4  import PointsForRendering
5
6  main = do
7      (progName,_) <- getArgsAndInitialize
8      createWindow progName
9      displayCallback $= display
10     reshapeCallback $= Just reshape
11     mainLoop
12
13 display = do
14     clear [ColorBuffer]
15     displayPoints points Quads
16     where
17         points
18             = [(0.5,0.5,0)
19                ,(-0.5,0.5,0)
20                ,(-0.5,-0.5,0)
21                ,(0.5,-0.5,0)]
22
23 reshape s@(Size w h) = do
24     viewport $= (Position 0 0, s)

```

If you start this program and resize the window, then always the complete window pane will be used for rendering your image.

Beispiel:

In this example only parts of the window are used for rendering the image. The image is smaller than the window.

```

1  import Graphics.UI.GLUT
2  import Graphics.Rendering.OpenGL
3
4  import PointsForRendering
5
6  main = do
7      (progName,_) <- getArgsAndInitialize
8      createWindow progName
9      clearColor $= Color4 0 0 0 0
10     displayCallback $= display
11     reshapeCallback $= Just reshape
12     mainLoop
13
14 display = do

```

```

15   clearColor $= Color4 1 1 1 1
16   clear [ColorBuffer]
17   currentColor $= Color4 1 0 0 1
18   displayPoints ps1 LineLoop
19   displayPoints ps2 Lines
20   where
21     ps1=[(0.5,0.5,0),(-0.5,0.5,0),(-0.5,-0.5,0),(0.5,-0.5,0)]
22     ps2=[(1,1,0),(-1,-1,0),(-1,1,0),(1,-1,0) ]
23
24   reshape s@(Size w h) = do
25     viewport $= (Position 50 50, Size (w-80) (h-60))

```

The resulting window can be found in figure 4.2.

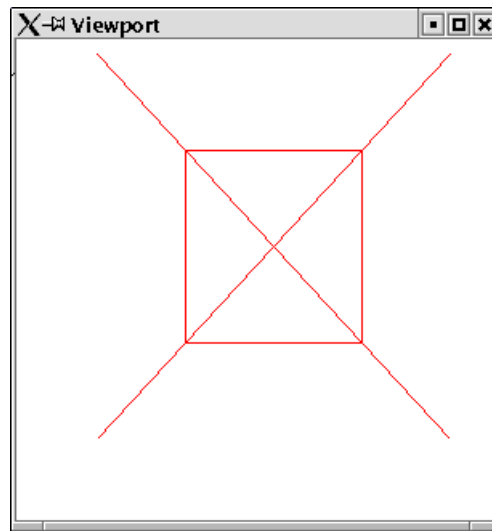


Figure 4.2: A Viewport smaller than the window.

4.3 Orthographic Projection

The viewport defines which parts of your window pane are used for rendering your image. The actual projection defines which coordinates you want to display. The simplest way to specify this is by the function `ortho`. It has six arguments, the lower and upper bounds of the x, y, z coordinates.

Projection is equally as transformation internally expressed in terms of a matrix. The statement `loadIdentity` can refer to the transformation or to the projection matrix. A state variable `matrixMode` defines, which of these matrixes these statements refer to. Therefore it is necessary to switch this variable to the value `Projection`, before applying the function `ortho` and afterwards to reset the variable back to the value `ModelView`.

Beispiel:

We render the same image in two windows with different projection values:

```

Ortho.hs
1 import PointsForRendering
2 import Graphics.Rendering.OpenGL
3 import Graphics.UI.GLUT as GLUT
4 import Star
5
6 main = do
7     (progName,_) <- getArgsAndInitialize
8
9     createWindow (progName++"1")
10    displayCallback $= display
11    projection (-5) 5 (-5) 5 (-5) 5
12
13    createWindow (progName++"2")
14    displayCallback $= display
15    projection 0 0.8 (-0.8) 0.8 (-0.5) 0.5
16
17    mainLoop
18
19 projection x1 xu y1 yu z1 zu = do
20     matrixMode $= Projection
21     loadIdentity
22     ortho x1 xu y1 yu z1 zu
23     matrixMode $= Modelview 0
24
25 display = do
26     clearColor $= Color4 1 1 1 1
27     clear [ColorBuffer]
28     currentColor $= Color4 1 0 0 1
29     star 0.9 7
30     currentColor $= Color4 1 1 0 1
31     star 0.4 5

```

The resulting windows can be found in figure 4.3.

`ortho` is the simplest projection we can define. When we will consider third dimensional szeneries we will learn a more powerful projection.

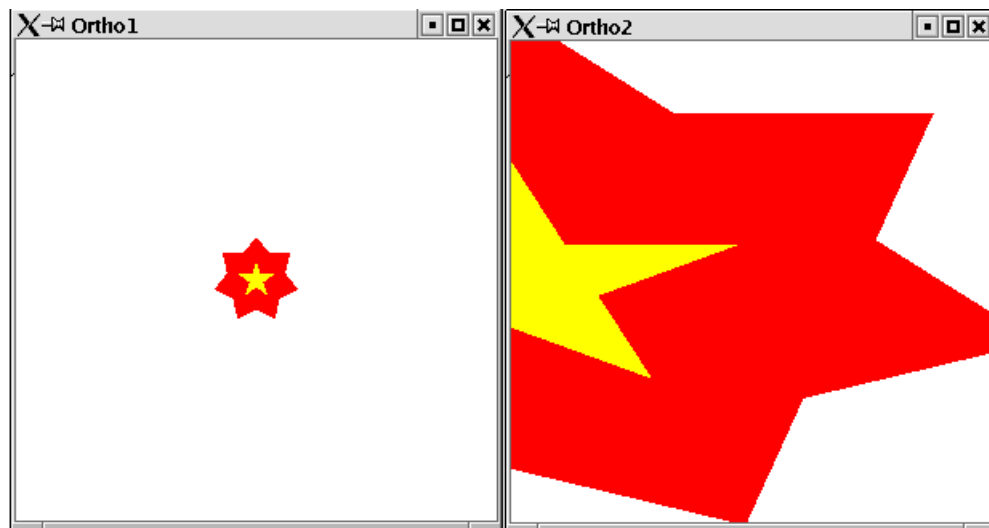


Figure 4.3: Two windows with different projection.

Chapter 5

Changing States

OpenGL is not only designed to render static images, but to have changing images. There are to ways how your image might change:

- it might react to some event, like some keyboard input or mouse event.
- it might change over time.

In order to change your image in some coordinated way, you need a state which can change. An event may change your state, or over the time your state might be changed.

5.1 Modelling your own State

A state is of course something, which does not match the purely functional paradigm of Haskell. However in the context of I/O the designers of Haskell came up with some clever way to integrate state changing variables into the Haskell's purely functional setting. The trick are again monads, as you have seen before for the state machine of OpenGL. There is a standard library in Haskell for state changing variables: `Data.IORef`. This provides functions for creation, setting, retrieving and modification of state variables. These functions are called:

`newIORef`, `writeIORef`, `readIORef`, `modifyIORef`.

If you think these names a bit too technical, then you might use the following module, which makes `IORef` variables instances of the type classes `HasGetter` and `HasSetter`. Thus we can use our own state variables in the same way, we use the `HOpenGL` state variables.

```
----- StateUtil.hs -----
1 module StateUtil where
2
3 import Graphics.Rendering.OpenGL
4 import Data.IORef
5 import Graphics.UI.GLUT
6
7 instance HasSetter IORef where
8     ($=) var val = writeIORef var val
```

```

9
10 instance HasGetter IORef where
11     get var = readIORef var
12
13 new = newIORef

```

5.2 Handling of Events

Now we know how to model our own state. We can use this for reacting on some events. Event handling in HOpenGL is done by setting a callback function for mouse and keyboard events. A callback function for mouse and keyboard events needs to be of the following type:

```

1 type KeyboardMouseCallback =
2     Key -> KeyState -> Modifiers -> Position -> IO ()

```

A Key can be some character, some special character or some mouse button:

```

1 data Key
2     = Char Char
3     | SpecialKey SpecialKey
4     | MouseButton MouseButton
5     deriving ( Eq, Ord, Show )

```

The keystate informs, if the key has been pressed or released.

```

1 data KeyState
2     = Down
3     | Up
4     deriving ( Eq, Ord, Show )

```

A modifier denotes, if some extra key is used, like the alt, strg or shift key:

```

1 data Modifiers = Modifiers { shift, ctrl, alt :: KeyState }
2     deriving ( Eq, Ord, Show )

```

And finally the position informs about the current mouse pointer position.

5.2.1 Keyboard events

With the close look at the event handling function above it is fairly easy to write a program that reacts on keyboard events. A function of type `KeyboardMouseCallback` is to be written and assigned to the state variable `keyboardMouseCallback` of your window. Usually your `KeyboardMouseCallback` will have access to some of your state variables, since you want to change a state when an event occurs. When the state has been changed, HOpenGL needs to be forced to redisplay the picture with the new state values. Therefore a call to the function `postRedisplay` needs to be done.

Beispiel:

In this example we draw a circle. The radius of the circle can be changed by use of the + and - key.

```

1  import Circle
2  import PointsForRendering
3  import StateUtil
4
5  import Graphics.Rendering.OpenGL
6  import Data.IORef
7  import Graphics.UI.GLUT
8
9  main = do
10     (progName,_) <- getArgsAndInitialize
11     createWindow progName

```

We create a state variable which stores the current radius of the circle:

```

12  radius <- new 0.1

```

The display function gets this state variable as first argument:

```

13  displayCallback $= display radius

```

And the keyboard callback gets this variable as first argument:

```

14  keyboardMouseCallback $= Just (keyboard radius)
15  mainLoop

```

The display function gets the current value for the radius and draws a filled circle:

```

16  display radius = do
17     clear [ColorBuffer]
18     r <- get radius
19     fillCircle r

```

The keyboard callback reacts on two keyboard events. The value of the radius variable are changed:

```

20  keyboard radius (Char '+') Down _ _ = do
21     r <- get radius
22     radius $= r+0.05
23     postRedisplay Nothing
24  keyboard radius (Char '-') Down _ _ = do
25     r <- get radius
26     radius $= r-0.05
27     postRedisplay Nothing
28  keyboard _ _ _ _ _ = return ()

```

Compile and start this program and press the + and - key.

5.3 Changing State over Time

The second way to change your picture is over time. You can create an animation if your picture changes a tiny bit every moment. In HOpenGL you can define a so called *idle* function. This function will be evaluated whenever the picture has been displayed. There you can define, in what way your state will change before the next redisplay is performed. The last statement in an *idle* function will be usually a call to `postRedisplay`.

Beispiel:

We define our first animation. A ring is displayed with a changing radius.

```

1  import Ring
2  import PointsForRendering
3  import StateUtil
4
5  import Graphics.Rendering.OpenGL
6  import Data.IORef
7  import Graphics.UI.GLUT as GLUT
Idle.hs

```

We define a constant which denotes the value by which the radius changes between every redisplay:

```

8
9  _STEP = 0.001
Idle.hs

```

Within the main function an *idle* callback is added to the window:

```

10 main = do
11     (progName,_) <- getArgsAndInitialize
12     createWindow progName
13     radius <- new 0.1
14     step <- new _STEP
15     displayCallback $= display radius
16     idleCallback $= Just (idle radius step)
17     mainLoop
Idle.hs

```

The display function renders a ring, depending on the state variable for the radius:

```

18 display radius = do
19     clear [ColorBuffer]
20     r <- get radius
21     ring r (r+0.2)
22     flush
Idle.hs

```

The idle function changes the value of the variable `radius` depending on the second state variable `step`.

```

23 idle radius step = do
24   r <- get radius
25   s <- get step
26   if r>=1 then step $= (-_STEP)
27       else if r<=0 then step $= _STEP
28                               else return ()
29   s <- get step
30   radius $= r+s
31   postRedisplay Nothing
Idle.hs

```

5.3.1 Double buffering

The animation created in the last example was not very satisfactory. A ring with changing radius was displayed, but the animation was somehow flickering. The reason for that was, that the display function as its first statement clears the screen, i.e. makes it altogether black. Only afterwards the ring is rendered. For a short moment the screen will be completely black. This is what makes this flickering effect.

A common solution for this problem in animated pictures is, not to apply the statements of the display function directly to the screen, but to an invisible buffer. When all statements of the display function have been applied to this invisible background buffer, this buffer is copied to the screen. This way only the ready to use final picture is shown on screen and not any intermediate rendering step (e.g. the picture after the clear statement).

OpenGL provides a double buffering mechanism. We only have to activate this. Therefore we need to set the initial display mode variable accordingly. Instead of a call to the function `flush` a call to the function `swapBuffers` needs to be done as last statement of the display function.

Beispiel:

The ring with changing radius over time now with double buffering.

```

1 import Ring
2 import PointsForRendering
3 import StateUtil
4
5 import Graphics.Rendering.OpenGL
6 import Data.IORef
7 import Graphics.UI.GLUT as GLUT
8
9 _STEP = 0.001
10
11 main = do
12   (progName,_) <- getArgsAndInitialize
13   initialDisplayMode $= [GLUT.Double]
14   createWindow progName
15   radius <- new 0.1
Double.hs

```

```

16   step    <- new _STEP
17   displayCallback $= display radius
18   idleCallback $= Just (idle radius step)
19   mainLoop
20
21   display radius = do
22     clear [ColorBuffer]
23     r <- get radius
24     ring r (r+0.2)
25     swapBuffers
26
27   idle radius step = do
28     r <- get radius
29     s <- get step
30     if r>=1 then step $= (-_STEP)
31         else if r<=0 then step $= _STEP
32                     else return ()
33
34     s <- get step
35     radius $= r+s
36     postRedisplay Nothing

```

5.4 Pong: A first Game

By now you have seen a lot of tiny examples. It is time to draw the techniques together and do an application with HOpenGL. In this section we will implement one of the first animated computer games ever: **Pong**. It consists of a small white circle which moves over a black screen and two paddles which can move on a vertical line.

Pong in action can be found in figure 5.1.

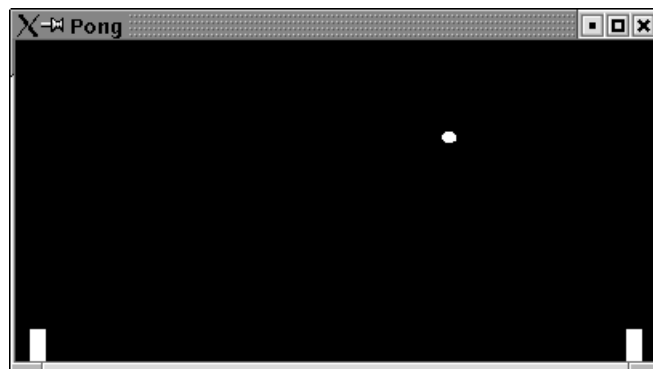


Figure 5.1: Pong in action.

```

1   import Circle
2   import Squares
3   import PointsForRendering

```

Pong.hs

```

4 import StateUtil
5
6 import Graphics.Rendering.OpenGL
7 import Data.IORef
8 import Graphics.UI.GLUT as GLUT

```

First of all we define some constant values for the game: x-, y-coordinates of the game, width and height of a paddle, the radius of the ball, initial factor, how a ball and a paddle changes its position, and an initial board size.

```

9      _____ Pong.hs _____
10  _LEFT  = -2
11  _RIGHT = 1
12  _TOP   = 1
13  _BOTTOM= -1
14
15  paddleWidth  = 0.07
16  paddleHeight = 0.2
17  ballRadius   = 0.035
18
19  _INITIAL_WIDTH :: GLsizei
20  _INITIAL_WIDTH=400
21
22  _INITIAL_HEIGHT::GLsizei
23  _INITIAL_HEIGHT=200
24
25  _INITIAL_BALL_DIR = 0.002
26  _INITIAL_PADDLE_DIR = 0.005

```

We define a data type, game. The game state can be characterized by the position of the ball and the values these coordinates change for the next redisplay:

```

26      _____ Pong.hs _____
27  data Ball    = Ball (GLfloat,GLfloat) GLfloat GLfloat

```

The paddles, which are characterized by their position and the position change on the y-axis (x-axis is fixed for a paddle).

```

27      _____ Pong.hs _____
28  type Paddle = (GLfloat,GLfloat,GLfloat)

```

Additionally a game has points for the left and the right player and a factor which denotes how fast ball and paddles move:

```

28      _____ Pong.hs _____
29  data Game
30  = Game { ball :: Ball
31          , leftP,rightP :: Paddle
32          , points ::(Int,Int)
33          , moveFactor::GLfloat}

```

For a starting game we provide the following initial game state:

```

34  initGame
35  = Game {ball=Ball (-0.8,0.3) _INITIAL_BALL_DIR _INITIAL_BALL_DIR
36          ,leftP=(_LEFT+paddleWidth,_BOTTOM,0)
37          ,rightP=(_RIGHT-2*paddleWidth,_BOTTOM,0)
38          ,points=(0,0)
39          ,moveFactor=1
40          }

```

The main function creates a double buffering window in fullscreen mode. An initial game state is created and passed to the keyboard, display, idle and reshape function:

```

41  main = do
42    (progName,_) <- getArgsAndInitialize
43    initialDisplayMode $= [GLUT.Double]
44    createWindow progName
45    game <- newIORef initGame
46    --windowSize $= Size _INITIAL_WIDTH _INITIAL_HEIGHT
47    fullScreen
48    displayCallback $= display game
49    idleCallback $= Just (idle game)
50    keyboardMouseCallback $= Just (keyboard game)
51    reshapeCallback $= Just (reshape game)
52    mainLoop

```

The display function simply gets the ball and paddles from the game state and renders these:

```

53  display game = do
54    clear [ColorBuffer]
55    g <- get game
56    let (Ball pos xDir yDir) = ball g
57    --a ball is a circle
58    displayAt pos $ fillCircle ballRadius
59    displayPaddle$leftP g
60    displayPaddle$rightP g
61    swapBuffers

```

Paddles are simply rectangles:

```

62  displayPaddle (x,y,_) = matrixExcursion$do
63    translate$Vector3 (paddleWidth/2) (paddleHeight/2) 0
64    displayAt (x,y)$myRect paddleWidth paddleHeight

```

We made use of the utility function which moves a shape to some position:


```

65 displayAt (x, y) displayMe = matrixExcursion$do
66   translate$Vector3 x y (0::GLfloat)
67   displayMe

```

Within the idle function ball and paddles need to be set to their next position on the field:

```

68 idle game = do
69   g <- get game
70   let fac = moveFactor g
71   game
72   $= g{ball    = moveBall g
73        ,leftP  = movePaddle (leftP g) fac
74        ,rightP = movePaddle (rightP g) fac
75        }
76   postRedisplay Nothing

```

The movement on the ball is determined by the upper and lower bound of the field, by the left and right bound of the field and the position of the paddles:

```

77 moveBall g
78 = Ball (x+factor*newXDir,y+factor*newYDir) newXDir newYDir
79 where
80   newXDir
81   | x-ballRadius <= xl+paddleWidth
82   && y+ballRadius >=yl
83   && y <=yl+paddleHeight
84   = -xDir
85   | x <= _LEFT-ballRadius = 0
86   | x+ballRadius >= xr
87   && y+ballRadius >=yr
88   && y <=yr+paddleHeight
89   = -xDir
90   | x >= _RIGHT+ballRadius = 0
91   | otherwise = xDir
92   newYDir
93   | y > _TOP-ballRadius || y< _BOTTOM+ballRadius = -yDir
94   | newXDir == 0 = 0
95   | otherwise = yDir
96   (Ball (x,y) xDir yDir) = ball g
97   factor = moveFactor g
98   (xl,yl,_) = leftP g
99   (xr,yr,_) = rightP g
100

```

A paddle moves only on the y-axis. We just need to ensure that it does not leaves the field. There are maximum and minimum values for y:

```

101 movePaddle (x,y,dir) factor =
102   let y1 = y+ factor*dir
103       newY = min (_TOP-paddleHeight) $max _BOTTOM y1
104   in (x,newY,dir)

```

The keyboard function: key 'a' moves the left paddle, key 'l' the right paddle and the space key gets a new ball:

```

105
106 keyboard game (Char 'a') upDown _ _ = do
107   g <- get game
108   let (x,y,_) = leftP g
109   game $= g{leftP=(x,y,paddleDir upDown)}
110 keyboard game (Char 'l') upDown _ _ = do
111   g <- get game
112   let (x,y,_) = rightP g
113   game $= g{rightP=(x,y,paddleDir upDown)}
114 keyboard game (Char '\32') Down _ _ = do
115   g <- get game
116   let Ball (x,y) xD yD = ball g
117   let xDir
118       | x<=_LEFT+3*paddleWidth = _INITIAL_BALL_DIR
119       | x>=_RIGHT-3*paddleWidth = - _INITIAL_BALL_DIR
120       | otherwise = xD
121   if (xD==0)
122     then game$=g{ball=Ball (x+4*xDir,y) xDir _INITIAL_BALL_DIR}
123     else return ()
124 keyboard _ _ _ _ _ = return ()
125
126 paddleDir Down = _INITIAL_PADDLE_DIR
127 paddleDir Up   = -_INITIAL_PADDLE_DIR

```

Finally we define the visual part of the screen. The movement factor of the ball depends on the width of the screen:

```

128 reshape game s@(Size w h) = do
129   viewport $= (Position 0 0, s)
130   matrixMode $= Projection
131   loadIdentity
132   ortho (-2.0) 1.0 (-1.0) 1.0 (-1.0) 1.0
133   matrixMode $= Modelview 0
134   g <- get game
135   game$=g{moveFactor=fromIntegral w/fromIntegral _INITIAL_WIDTH}

```

Have a break and play *Pong*.

Chapter 6

Third Dimension

Up to now everything was pretty boring. We never considered the three dimensional space provided by OpenGL. Strictly we just considered two dimensions. Thus the library was not any more powerfull than any simple graphics libaray e.g. like Java's `java.awt.Graphics` class. In this chapter we will explore the true power of OpenGL by actually rendering three dimensional objects.

6.1 Hidden Shapes

In a three dimensional space some objects will be in front of others and hide them. We would expect to see only those areas which are not hidden by areas closer to the viewer.

Beispiel:

We render two shapes. A red square which is closer to the viewer and a blue circle which is farer away:

```

1  import Graphics.Rendering.OpenGL
2  import Graphics.UI.GLUT as GLUT
3  import Squares
4  import Circle
5  import PointsForRendering
6
7  main = do
8      (progName,_) <- getArgsAndInitialize
9      createWindow progName
10     displayCallback $= display
11     clearColor $= Color4 1 1 1 1
12     mainLoop
13
14 display = do
15     clear [ColorBuffer,DepthBuffer]
16     loadIdentity
17     translate (Vector3 0 0 (-0.5::GLfloat))

```

```

18   currentColor $= Color4 1 0 0 1
19   square 1
20
21   loadIdentity
22   translate (Vector3 0.2 0.2 (0.5::GLfloat))
23   currentColor $= Color4 0 0 1 1
24   fillCircle 0.5
25   flush

```

However as can be seen in figure 6.1, the blue circle hides parts of the red square.

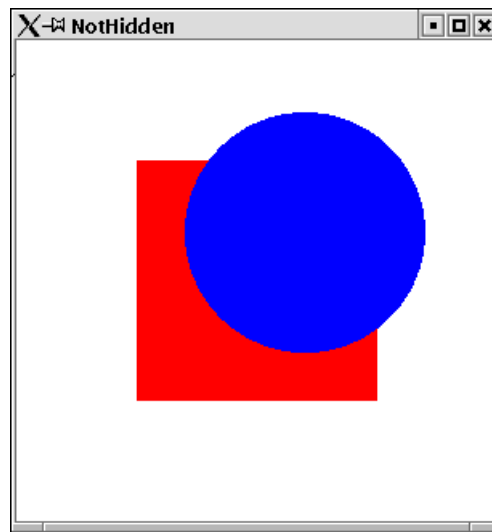


Figure 6.1: Third dimension not correctly taken into account.

By default OpenGL does not take the depth into account. Shapes rendered later hide other shapes which were rendered earlier, neglecting the depth of the shapes. OpenGL provides a mechanism for automatically considering the depth of a shape. This simply needs to be activated. Three steps need to be done:

- as initial display mode **Depth** needs to be set.
- a depth function needs to be set. Usually the **Less** mode is used as function here. This ensures that closer objects hide objects farer away.
- the depth buffer needs to be cleared in the beginning of the display function.

Beispiel:

Now we render the same to shapes as in the example before, but the depth mechanism of OpenGL is activated.

```

Hidden.hs
1   import Graphics.Rendering.OpenGL
2   import Graphics.UI.GLUT    as GLUT
3

```

```

4 import Squares
5 import Circle
6 import PointsForRendering
7
8 main = do
9     (progName,_) <- getArgsAndInitialize
10    initialDisplayMode $= [Depth]
11
12    createWindow progName
13
14    depthFunc $= Just Less
15    displayCallback $= display
16    clearColor $= Color4 1 1 1 1
17    mainLoop
18
19 display = do
20    clear [ColorBuffer,DepthBuffer]
21    loadIdentity
22    translate (Vector3 0 0 (-0.5::GLfloat))
23    currentColor $= Color4 1 0 0 1
24    square 1
25
26    loadIdentity
27    translate (Vector3 0.2 0.2 (0.5::GLfloat))
28    currentColor $= Color4 0 0 1 1
29    fillCircle 0.5
30    flush

```

Now as can be seen in figure 6.2, the red square hides parts of the blue circle.

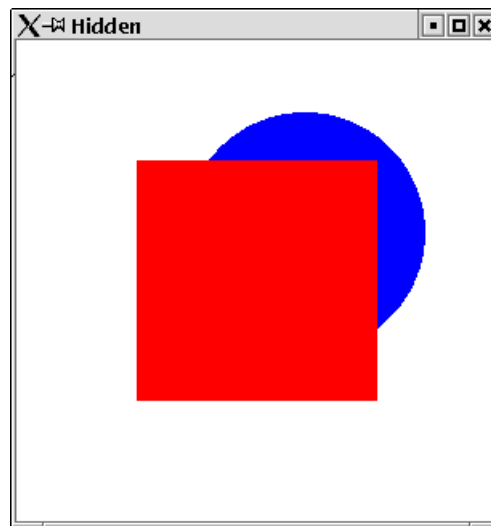


Figure 6.2: Third dimension correctly taken into account by use of depth function.

6.2 Perspective Projection

In the real world objects closer to the viewer appear larger than objects farther away from the viewer. Up to now we only learnt how to set up an orthographic projection. In an orthographic projection objects farther away have the same size as object close to the viewer.

Beispiel:

We can test the orthographic projection. Two squares equally in size, but in different distances from the viewer are rendered:

```

1 import Graphics.Rendering.OpenGL
2 import Graphics.UI.GLUT as GLUT
3
4 import Squares
5 import Circle
6 import PointsForRendering
7
8 main = do
9     (progName,_) <- getArgsAndInitialize
10    initialDisplayMode $= [Depth]
11    createWindow progName
12    depthFunc $= Just Less
13    displayCallback $= display
14
15    matrixMode $= Projection
16    loadIdentity
17    ortho (-5) 5 (-5) 5 (1) 40
18    matrixMode $= Modelview 0
19
20    clearColor $= Color4 1 1 1 1
21    mainLoop
22
23 display = do
24    clear [ColorBuffer,DepthBuffer]
25    loadIdentity
26    translate (Vector3 0 0 (-2::GLfloat))
27    currentColor $= Color4 1 0 0 1
28    square 1
29
30    loadIdentity
31    translate (Vector3 4 4 (-5::GLfloat))
32    currentColor $= Color4 0 0 1 1
33    square 1
34    flush

```

As can be seen in figure 6.3 , the two squares have the same size, even though the red one is closer to the viewer.

OpenGL provides the function `frustum` for specifying a perspective projection. `frustum` has 6 arguments:

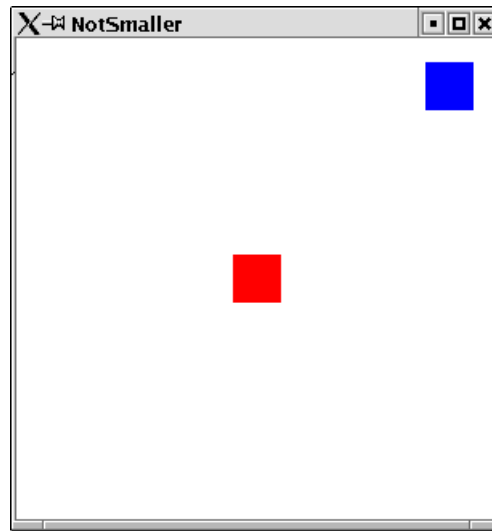


Figure 6.3: Two squares in orthographic projection.

- *left*: left bound for the closest orthogonal plane
- *right*: right bound for the closest orthogonal plane
- *top*: upper bound for the closest orthogonal plane
- *bottom*: lower bound for the closest orthogonal plane
- *near*: the closest things that can be seen
- *far*: the farthest away things that can be seen

Figure 6.4 illustrates these six values.

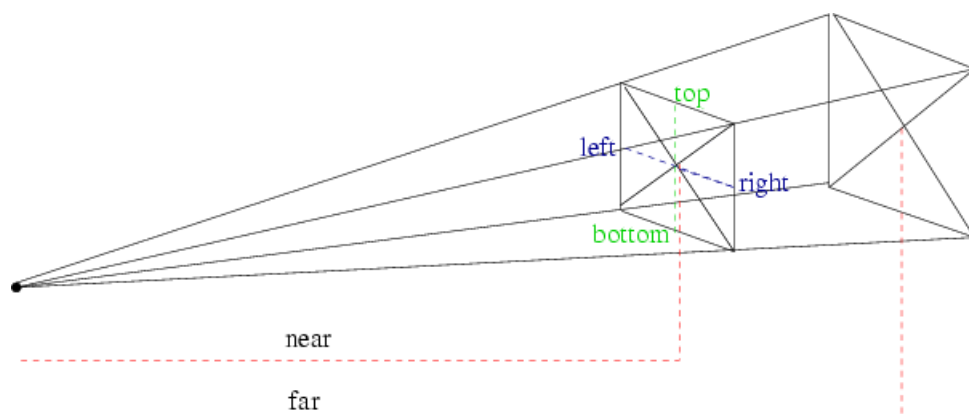


Figure 6.4: Perspective projection with frustum.

Usually you will have negated values for top/bottom and left/right.

Beispiel:

Now we render the two squares from the previous example again. This time we use a perspective projection:

```

1  import Graphics.Rendering.OpenGL
2  import Graphics.UI.GLUT as GLUT
3
4  import Squares
5  import Circle
6  import PointsForRendering
7
8  main = do
9      (progName,_) <- getArgsAndInitialize
10     initialDisplayMode $= [Depth]
11     createWindow progName
12     depthFunc $= Just Less
13     displayCallback $= display
14
15     matrixMode $= Projection
16     loadIdentity
17     let near    = 1
18         far     = 40
19         right   = 1
20         top     = 1
21     frustum (-right) right (-top) top near far
22     matrixMode $= Modelview 0
23
24     clearColor $= Color4 1 1 1 1
25     mainLoop
26
27 display = do
28     clear [ColorBuffer,DepthBuffer]
29     loadIdentity
30     translate (Vector3 0 0 (-2::GLfloat))
31     currentColor $= Color4 1 0 0 1
32     square 1
33
34     loadIdentity
35     translate (Vector3 4 4 (-5::GLfloat))
36     currentColor $= Color4 0 0 1 1
37     square 1
38     flush

```

Now, as can be seen in figure 6.5, the blue square appears to be smaller than the red square.

HOpenGL provides a second function to define a perspective projection: `perspective`. Here instead of left, right, top, bottom an angle between the top/bottom ray and the width of the closest plane can be specified.

Figure 6.6 illustrates these values.

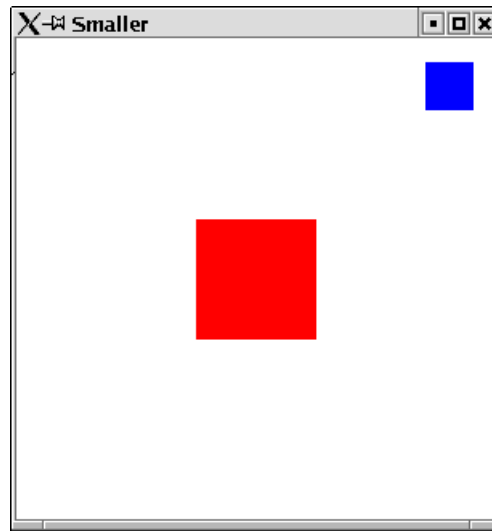


Figure 6.5: Two squares in perspective projection.

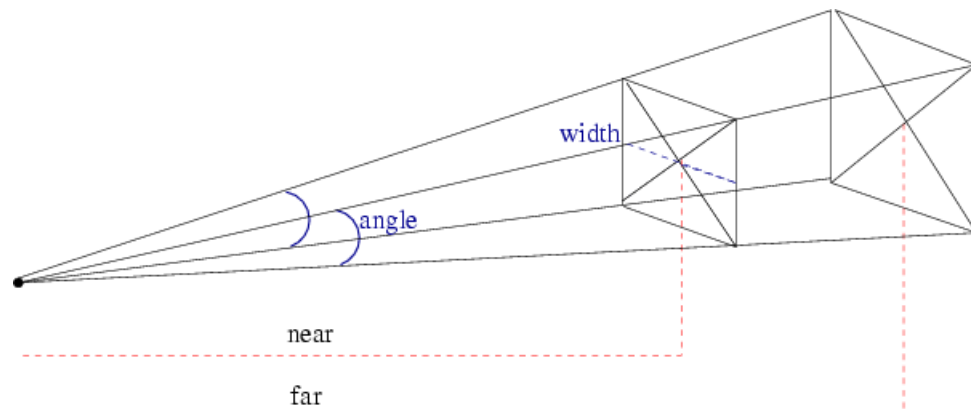


Figure 6.6: Perspective projection.

6.3 Setting up the Point of View

In the previous section we have learnt that there is a second way how to project the three dimensional space onto the two dimensional area of the screen. We did however not yet specify, where in the three dimensional space the viewer is situated and in what direction they are looking. In order to define this, OpenGL provides the function `lookAt`. It has three arguments:

- the point, where the viewer is situated.
- the point at which the viewer is looking.
- and a vector, which specifies the direction which is to be up for the viewer.

6.3.1 Orbiting around the origin

The point of view, where we are looking from is interesting, when we change it. In the following a module is defined, which allows the viewer to move along a sphere. The point of view can be set for a given sphere position. The position is specified by two angles and a radius. The first angle defines which way to move around the x-axis the second angle, which angle to move around the y-axis. The radius defines the distance from the origin. The position can be changed through keyboard events.

```

1 module OrbitPointOfView where
2
3 import Graphics.Rendering.OpenGL
4 import Graphics.UI.GLUT as GLUT
5
6 import StateUtil
7 import Data.IORef
8
9 setPointOfView pPos = do
10   (alpha,beta,r) <- get pPos
11   let
12     (x,y,z)      = calculatePointOfView alpha beta r
13     (x2,y2,z2) = calculatePointOfView ((alpha+90)'mod' 360) beta r
14     lookAt (Vertex3 x y z) (Vertex3 0 0 0) (Vector3 x2 y2 z2)
15
16 calculatePointOfView alp bet r =
17   let alpha = fromIntegral alp*2*pi/fromIntegral 360
18       beta  = fromIntegral bet*2*pi/fromIntegral 360
19       y = r * cos alpha
20       u = r * sin alpha
21       x = u * cos beta
22       z = u * sin beta
23   in (x,y,z)
24
25 keyForPos pPos (Char '+')      = modPos pPos (id,id,\x->x-0.1)
26 keyForPos pPos (Char '-')      = modPos pPos (id,id,(+)0.1)
27 keyForPos pPos (SpecialKey KeyLeft) = modPos pPos (id,(+)359,id)
28 keyForPos pPos (SpecialKey KeyRight)= modPos pPos (id,(+)1,id)
29 keyForPos pPos (SpecialKey KeyUp)   = modPos pPos ((+)1,id,id)
30 keyForPos pPos (SpecialKey KeyDown) = modPos pPos ((+)359,id,id)
31 keyForPos _ _                     = return ()
32
33 modPos pPos (ffst,fsnd,ftrd) = do
34   (alpha,beta,r) <- get pPos
35   pPos $= (ffst alpha 'mod' 360,fsnd beta 'mod' 360,ftrd r)
36   postRedisplay Nothing
37
38 reshape screenSize@(Size w h) = do
39   viewport $= ((Position 0 0), screenSize)
40   matrixMode $= Projection
41   loadIdentity

```

```

42   let near    = 0.001
43       far     = 40
44       fov     = 90
45       ang     = (fov*pi)/(360)
46       top     = near / ( cos(ang) / sin(ang) )
47       aspect  = fromIntegral(w)/fromIntegral(h)
48       right   = top*aspect
49   frustum (-right) right (-top) top near far
50   matrixMode $= Modelview 0

```

Beispiel:

Let us use the module above, to orbit around a cube. Therefore we define a simple module, which renders a cube with differently colored areas. The cube is situated at the origin. We render the six areas by rendering a square at the origin and translate and rotate it into its final position.

```

ColorCube.hs
1  module ColorCube where
2
3  import Graphics.Rendering.OpenGL
4  import Graphics.UI.GLUT    as GLUT
5
6  import Squares
7  import StateUtil
8
9  locally = matrixExcursion
10
11 colorCube n = do
12   locally $ do
13     currentColor $= Color4 1 0 0 1
14     translate$Vector3 0 0 (-n/2)
15     square n
16   locally $ do
17     currentColor $= Color4 0 1 0 1
18     translate$Vector3 0 0 (n/2)
19     square n
20   locally $ do
21     currentColor $= Color4 0 0 1 1
22     translate$Vector3 (n/2) 0 0
23     rotate 90 $Vector3 0 (1::GLfloat) 0
24     square n
25   locally $ do
26     currentColor $= Color4 1 1 0 1
27     translate$Vector3 (-n/2) 0 0
28     rotate 90 $Vector3 0 (1::GLfloat) 0
29     square n
30   locally $ do
31     currentColor $= Color4 0 1 1 1
32     translate$Vector3 0 (-n/2) 0
33     rotate 90 $Vector3 (1::GLfloat) 0 0

```

```

34     square n
35   locally $ do
36     currentColor $= Color4 1 1 1 1
37     translate$Vector3 0 (n/2) 0
38     rotate 90 $Vector3 (1::GLfloat) 0 0
39     square n

```

The following program allows to use the cursor keys to move around a cube at the origin:

```

----- OrbitAroundCube.hs -----
1  import Graphics.Rendering.OpenGL
2  import Graphics.UI.GLUT as GLUT
3
4  import Squares
5  import OrbitPointOfView
6  import StateUtil
7  import ColorCube
8
9  main = do
10     (progName,_) <- getArgsAndInitialize
11     initialDisplayMode $= [Depth,GLUT.Double]
12     createWindow progName
13     depthFunc $= Just Less
14
15     pPos <- new (90::Int,270::Int,2.0)
16     keyboardMouseCallback $= Just (keyboard pPos)
17
18     displayCallback $= display pPos
19     reshapeCallback $= Just reshape
20     mainLoop

```

The display function sets the viewer's position before rendering the cube:

```

----- OrbitAroundCube.hs -----
21 display pPos = do
22     loadIdentity
23     setPointOfView pPos
24     clear [ColorBuffer,DepthBuffer]
25     colorCube 1
26     swapBuffers

```

As keyboard function we map directly to the function defined in `OrbitPointOfView`.

```

----- OrbitAroundCube.hs -----
27 keyboard pPos c _ _ _ = keyForPos pPos c

```

An example how the colored cube can now be seen is given in figure 6.7.

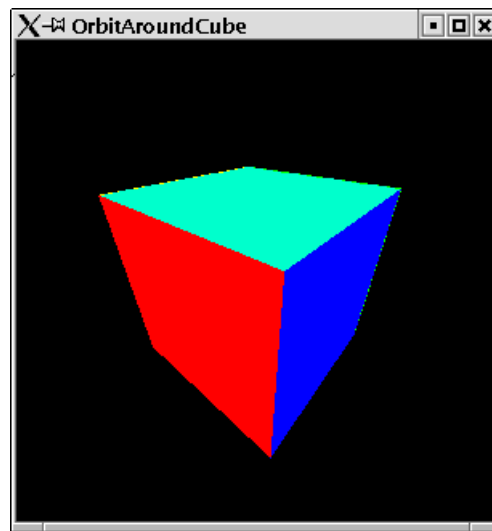


Figure 6.7: A view of the colored cube,

6.4 3D Game: Rubik's Cube

In this section we implement a primitive version of Rubik's cube. Rubik's cube in action can be found in figure 6.8.

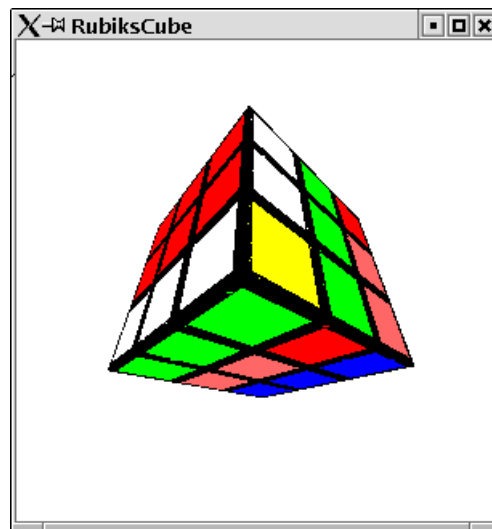


Figure 6.8: Rubik's Cube in action.

6.4.1 Cube Logics

First of all we model the logics of Rubik's Cube¹.

A data type is provided for representation of a cube:

```

1  module RubikLogic where
2
3  data Rubik a
4    = Rubik (Front a) (Top a) (Back a) (Bottom a) (Left a) (Right a)
5
6  type Front a  = Area a
7  type Top a    = Area a
8  type Back a   = Area a
9  type Bottom a = Area a
10 type Left a   = Area a
11 type Right a  = Area a
12
13 type Area a = [Row a]
14 type Row a  = [a]
15
16 data AreaPosition = Front | Top | Back | Bottom | Left | Right
17
18 data RubikColor = Red | Blue | Yellow | Green | Orange | White | Black

```

We make the type `Rubik` an instance of the class `Functor`:

```

19 instance Functor Rubik where
20   fmap f (Rubik front top back bottom left right)
21     = Rubik (mf front) (mf top) (mf back)
22             (mf bottom) (mf left) (mf right)
23   where
24     mf = map (map f)

```

The initial cube is defined

```

25 initCube = Rubik (area Red) (area Blue) (area Yellow)
26               (area Green) (area Orange) (area White)
27
28 area c = [[c,c,c],[c,c,c],[c,c,c]]

```

The main operation on a cube is to turn one of its six sides. The function `rotateArea` specifies, how this effects a cube.

¹There are certainly cleverer ways to do this, but I did not take the time to think of them.

```

29 rotateArea RubikLogic.Front
30   (Rubik front top back bottom left right) =
31     Rubik front' top' back bottom' left' right'
32     where
33       top'      = newRow 3 (reverse$column 3 left) top
34       bottom'   = newRow 1 (reverse$column 1 right) bottom
35       left'     = newColumn 3 (row 1 bottom) left
36       right'    = newColumn 1 (row 3 top) right
37       front'    = rotateBy3 front
38
39 rotateArea RubikLogic.Back
40   (Rubik front top back bottom left right) =
41     Rubik front' top' back' bottom'
42     (rotateBy2 left') (rotateBy2 right')
43     where
44       (Rubik back' bottom' front' top' left' right') =
45         rotateArea RubikLogic.Front
46         (Rubik back bottom front top
47          (rotateBy2 left) (rotateBy2 right))
48
49 rotateArea RubikLogic.Bottom
50   (Rubik front top back bottom left right) =
51     Rubik front' top' back' bottom' left' right'
52     where
53       back'     = newRow 1 (reverse$row 3 left) back
54       front'    = newRow 3 (row 3 right) front
55       left'     = newRow 3 (row 3 front) left
56       right'    = newRow 3 (reverse$row 1 back) right
57       bottom'   = rotateBy1 bottom
58
59
60 rotateArea RubikLogic.Top
61   (Rubik front top back bottom left right) =
62     Rubik front' top' back' bottom left' right'
63     where
64       back'     = newRow 3 (reverse$row 1 right) back
65       front'    = newRow 1 (row 1 left) front
66       left'     = newRow 1 (reverse$row 3 back) left
67       right'    = newRow 1 (row 1 front) right
68       top'     = rotateBy1 top
69
70 rotateArea RubikLogic.Left
71   (Rubik front top back bottom left right) =
72     Rubik front' top' back' bottom' left' right
73     where
74       top'      = newColumn 1 (column 1 front) top
75       bottom'   = newColumn 1 (column 1 back) bottom
76       left'     = rotateBy3 left
77       back'     = newColumn 1 (column 1 top) back

```

```

78     front' = newColumn 1 (column 1 bottom) front
79
80 rotateArea RubikLogic.Right
81 (Rubik front top back bottom left right) =
82   Rubik front' top' back' bottom' left right'
83   where
84     top'      = newColumn 3 (column 3 back) top
85     bottom'   = newColumn 3 (column 3 front) bottom
86     right'    = rotateBy3 right
87     back'     = newColumn 3 (column 3 bottom) back
88     front'    = newColumn 3 (column 3 top) front
89
90 rotateBy1
91   [[x1,x2,x3]
92    ,[x8,x,x4]
93    ,[x7,x6,x5]] =
94   [[x3,x4,x5]
95    ,[x2,x,x6]
96    ,[x1,x8,x7]]
97
98 rotateBy2 = rotateBy1 .rotateBy1
99 rotateBy3 = rotateBy2 .rotateBy1

```

Finally some useful functions for manipulation of an area are given.

```

----- RubikLogic.hs -----
100 column n  = map (\row->row !! (n-1))
101
102 row n area = area !!(n-1)
103
104 newRow 1 row [a,r,ea] = [row,r,ea]
105 newRow 2 row [a,r,ea] = [a,row,ea]
106 newRow 3 row [a,r,ea] = [a,r,row]
107
108 newColumn n column area = map (doIt n) areaC
109   where
110     areaC = zip area column
111     doIt 1 ((r:ow),c) = c:ow
112     doIt 2 ((r:o:w),c) = r:c:w
113     doIt 3 ((r:o:w:xs),c) = r:o:c:xs

```

6.4.2 Rendering the Cube

We have a logical modell of a cube. Now we can render this in a coordinate system. In an earlier section we allready provided a function to render one single side. We simply need to render the six sides and move them to the correct position.

```

----- RenderRubik.hs -----
1 module RenderRubik where
2

```



```

3 | import Graphics.Rendering.OpenGL as OpenGL hiding (Red,Green,Blue)
4 | import Graphics.UI.GLUT as GLUT hiding (Red,Green,Blue)
5 |
6 | import PointsForRendering
7 | import RubikLogic
8 | import RubikFace
9 | import Squares
10 |
11 | _FIELD_WIDTHTH :: GLfloat
12 | _FIELD_WIDTHTH = 1/3
13 |
14 | renderCube (Rubik front top back bottom left right) = do
15 |   render RubikLogic.Top      top
16 |   render RubikLogic.Back     back
17 |   render RubikLogic.Front    front
18 |   render RubikLogic.Bottom   bottom
19 |   render RubikLogic.Left     left
20 |   render RubikLogic.Right    right
21 |
22 | render Top      cs = matrixExcursion$do
23 |   translate $Vector3 (1.5*_FIELD_WIDTHTH) 0 0
24 |   rotate (90)$Vector3 0 1 (0::GLfloat)
25 |   renderCubeSide cs
26 |
27 | render RubikLogic.Back cs = matrixExcursion$ do
28 |   translate $Vector3 0 0 (-1.5*_FIELD_WIDTHTH)
29 |   rotate (180)$Vector3 0 0 (1::GLfloat)
30 |   rotate (180)$Vector3 1 0 (0::GLfloat)
31 |   renderCubeSide cs
32 |
33 | render Bottom cs = matrixExcursion$ do
34 |   translate $Vector3 (-1.5*_FIELD_WIDTHTH) 0 0
35 |   rotate (270)$Vector3 0 1 (0::GLfloat)
36 |   renderCubeSide cs
37 |
38 | render RubikLogic.Front cs = matrixExcursion$ do
39 |   translate $Vector3 0 0 (1.5*_FIELD_WIDTHTH)
40 |   renderCubeSide cs
41 |
42 | render RubikLogic.Left cs = matrixExcursion$ do
43 |   translate $Vector3 0 (1.5*_FIELD_WIDTHTH) 0
44 |   rotate (270) $Vector3 1 0 (0::GLfloat)
45 |   renderCubeSide cs
46 |
47 | render RubikLogic.Right cs = matrixExcursion$ do
48 |   translate $Vector3 0 (-1.5*_FIELD_WIDTHTH) 0
49 |   rotate (270) $Vector3 1 0 (0::GLfloat)
50 |   rotate (180)$Vector3 1 0 (0::GLfloat)
51 |   renderCubeSide cs
52 |

```

```

53 renderCubeSide css = renderArea _FIELD_WIDTH css
54
55 field = square _FIELD_WIDTH

```

The following function maps our abstract color type to concrete OpenGL colors:

```

RenderRubik.hs
56 doColor Red      = Color4 1 0 0 1.0
57 doColor Green    = Color4 0 1 0 1.0
58 doColor Blue     = Color4 0 0 1 1.0
59 doColor Yellow   = Color4 1 1 0 1.0
60 doColor Orange   = Color4 1 0.5 0.5 1
61 doColor White    = Color4 1 1 1 1.0
62 doColor Black    = Color4 0 0 0 1.0

```

6.4.3 Rubik's Cube

Finally we can create a simple application.

```

RubiksCube.hs
1 import Graphics.Rendering.OpenGL
2 import Graphics.UI.GLUT as GLUT
3
4 import Data.IORef
5
6 import OrbitPointOfView
7 import StateUtil
8 import RubikLogic
9 import RenderRubik
10
11 main = do
12     initialDisplayMode $= [GLUT.Double, GLUT.RGB, Depth]
13     (progName, _) <- getArgsAndInitialize
14
15     createWindow progName
16
17     depthFunc $= Just Less
18
19     pPos <- new (90::Int, 270::Int, 2.0)
20     pCube <- new initCube
21
22     displayCallback $= display pPos pCube
23     keyboardMouseCallback $= Just (keyboard pPos pCube)
24
25     reshapeCallback $= Just reshape
26     mainLoop
27
28 display pPos pCube = do
29     clearColor $= Color4 1 1 1 1

```

```

30  clear [ColorBuffer,DepthBuffer]
31  loadIdentity
32  setPointOfView pPos
33  cube <- get pCube
34  renderCube$fmmap doColor cube
35  swapBuffers
36
37  keyboard _ pCube (Char '1') Down _ _
38      = rot pCube RubikLogic.Top
39  keyboard _ pCube (Char '2') Down _ _
40      = rot pCube RubikLogic.Bottom
41  keyboard _ pCube (Char '3') Down _ _
42      = rot pCube RubikLogic.Front
43  keyboard _ pCube (Char '4') Down _ _
44      = rot pCube RubikLogic.Back
45  keyboard _ pCube (Char '5') Down _ _
46      = rot pCube RubikLogic.Left
47  keyboard _ pCube (Char '6') Down _ _
48      = rot pCube RubikLogic.Right
49  keyboard pPos _ c _ _
50      = keyForPos pPos c
51
52  rot pCube p = do
53      cube <- get pCube
54      pCube $= rotateArea p cube
55      postRedisplay Nothing

```

6.5 Light

Let us begin with a simple 3-dimensional shape: a cube. A cube has six squares which we can render as the primitive shape `Quad`.

```

1  module Cube where
2
3  import Graphics.Rendering.OpenGL
4  import Graphics.UI.GLUT as GLUT
5
6  import PointsForRendering
7
8  cube l = renderAs Quads corners
9      where
10         corners =
11             [(1,0,1),(0,0,1),(0,1,1),(1,1,1)
12              ,(1,1,1),(1,1,0),(1,0,0),(1,0,1)
13              ,(0,0,0),(1,0,0),(1,0,1),(0,0,1)
14              ,(1,1,0),(0,1,0),(0,0,0),(1,0,0)
15              ,(0,1,1),(1,1,1),(1,1,0),(0,1,0)

```

```
16     ,(0,1,1),(0,1,0),(0,0,0),(0,0,1)
17 ]
```

Beispiel:

We make a first try at rendering a cube:

```
RenderCube.hs
1 import Graphics.Rendering.OpenGL
2 import Graphics.UI.GLUT as GLUT
3
4 import Cube
5
6 main = do
7     (progName,_) <- getArgsAndInitialize
8     createWindow progName
9     displayCallback $= display
10    mainLoop
11
12 display = do
13     clear [ColorBuffer]
14     rotate 40 (Vector3 1 1 (1::GLfloat))
15     cube 0.5
16     loadIdentity
17     flush
```

The resulting window can be found in figure 6.9. It is not very exiting, we see a white shape, which has the outline of a cube, but do not get the three dimensional visual effect of a cube.

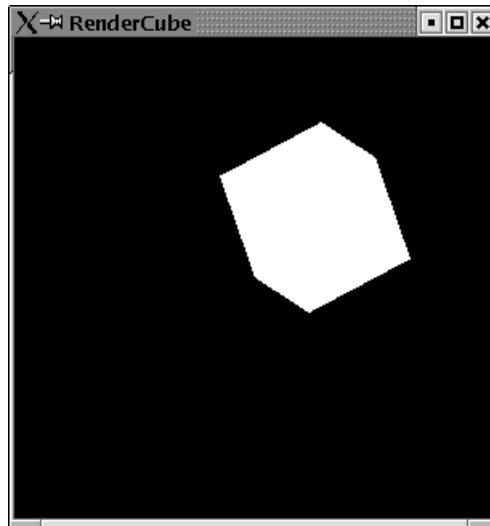


Figure 6.9: An unlit cube.

6.5.1 Defining a light source

For rendering three dimensional objects it is not enough to specify their shapes and your viewing position. Crucial is the way the objects are illuminated. In order to get a three dimensional visual effect on your two dimensional computer screen, it needs to be defined what kind of light source lights the object.

A light source can be specified fairly easy. First you need to set the state variable `lighting` to the value `Enabled`. Then you need to specify the position of your light source. This can be done by setting a special position state variable, e.g. by

```
position (Light 0) $= Vertex4 0.8 0 3.0 5.0.
```

And finally you need to turn the light source on by setting its state variable to enabled: `light (Light 0) $= Enabled`.

Beispiel:

Now we can render a cube with a defined light source:

```

1  import Graphics.Rendering.OpenGL
2  import Graphics.UI.GLUT as GLUT
3
4  import Cube
5
6  main = do
7      (progName,_) <- getArgsAndInitialize
8
9      depthFunc $= Just Less
10
11     createWindow progName
12
13     lighting $= Enabled
14     position (Light 0) $= Vertex4 1 0.4 0.8 1
15     light (Light 0) $= Enabled
16
17     displayCallback $= display
18     mainLoop
19
20 display = do
21     clear [ColorBuffer]
22     rotate 40 (Vector3 1 1 (1::GLfloat))
23     cube 0.5
24     loadIdentity
25     flush

```

The resulting window can be found in figure 6.10. Now we can identify a bit more the cube.

You might wonder, why the vertex for the light source position has four parameters. The forth parameter is a value by which the other three (the x, y, z coordinates) get divided.

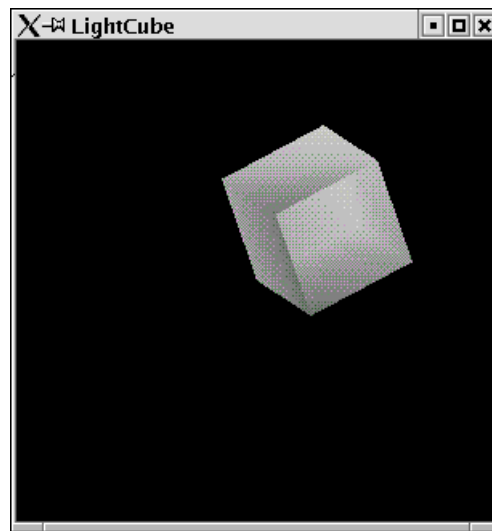


Figure 6.10: A lit cube.

6.5.2 Tux the Penguin

Let us render some cute object: Tux the penguin. We will roughly use the data from the OpenGL game tuxracer. The nice thing about a penguin is, that you can built it almost completely out of spheres. We will render Tux simply by rendering spheres, which are scaled to different forms and moved to the correct position.

Overall Setup

```

1  import Graphics.Rendering.OpenGL as OpenGL
2  import Graphics.UI.GLUT as GLUT
3
4  import OrbitPointOfView
5  import StateUtil
6
7  main = do
8      (progName,_) <- getArgsAndInitialize
9      initialDisplayMode $= [Depth,GLUT.Double]
10     pPos <- new (90::Int,270::Int,1.0)
11     depthFunc $= Just Less
12     createWindow progName
13
14     lighting $= Enabled
15     normalize $= Enabled
16     depthFunc $= Just Less
17
18     position (Light 0) $= Vertex4 0 0 (10) 0
19     ambient (Light 0) $= Color4 1 1 1 1
20     diffuse (Light 0) $= Color4 1 1 1 1

```

```

21     specular (Light 0) $= Color4 1 1 1 1
22     light (Light 0) $= Enabled
23
24     displayCallback $= display pPos
25     keyboardMouseCallback $= Just (keyboard pPos)
26     reshapeCallback $= Just reshape
27     mainLoop
28
29 keyboard pPos c _ _ _ = keyForPos pPos c

```

The main display function clears the necessary buffers and calls the main function for rendering the penguin Tux.

```

----- Tux.hs -----
30 display pPos = do
31     loadIdentity
32     clearColor $= Color4 1 0 0 1
33     setPointOfView pPos
34     clear [ColorBuffer,DepthBuffer]
35     tux
36     swapBuffers

```

Auxilliary Functions

We will use some auxilliary functions. First of all a function, which renders a scaled sphere.

```

----- Tux.hs -----
37 sphere r xs ys zs = do
38     scal xs ys zs
39     createSphere r
40
41 createSphere r = renderObject Solid $GLUT.Sphere r 50 50
42
43 scal:: GLfloat -> GLfloat -> GLfloat -> IO ()
44 scal x y z = scale x y z

```

Furthermore some functions for easy translate and rotate transformations:

```

----- Tux.hs -----
45 transl:: GLfloat -> GLfloat -> GLfloat -> IO ()
46 transl = translate.Vector3
47
48 rota:: GLfloat -> GLfloat -> GLfloat -> GLfloat -> IO ()
49 rota a = rotate a . Vector3
50
51 rotateZ a = rota a 0 0 1
52 rotateY a = rota a 0 1 0
53 rotateX a = rota a 1 0 0
54

```

```

55 |scal:: GLfloat -> GLfloat -> GLfloat -> IO ()
56 |scal = scale

```

And eventually some functions to set the material properties for the different parts of a penguin.

```

                                     Tux.hs
57 |crMat (rd,gd,bd) (rs,gs,bs) exp = do
58 |    materialDiffuse   Front $= Color4 rd gd bd  1.0
59 |    materialAmbient   Front $= Color4 rd gd bd  1.0
60 |    materialSpecular  Front $= Color4 rs gs bs  1.0
61 |    materialShininess Front $= exp
62 |
63 |    materialDiffuse   Back  $= Color4 rd gd bd  1.0
64 |    materialSpecular  Back  $= Color4 rs gs bs  1.0
65 |    materialShininess Back  $= exp
66 |
67 |whitePenguin = crMat (0.58, 0.58, 0.58)(0.2, 0.2, 0.2) 50.0
68 |blackPenguin = crMat (0.1, 0.1, 0.1)  (0.5, 0.5, 0.5) 20.0
69 |beakColour   = crMat (0.64, 0.54, 0.06)(0.4, 0.4, 0.4) 5
70 |nostrilColour= crMat (0.48039, 0.318627, 0.033725)(0.0,0.0,0.0) 1
71 |irisColour   = crMat (0.01, 0.01, 0.01)(0.4, 0.4, 0.4) 90.0

```

Torso and Head

The neck and torso of a penguin are almost black spheres with some white front parts. We will model such figures by setting a white sphere in front of a black sphere.

```

                                     Tux.hs
72 |makeBody = do
73 |    matrixExcursion$do
74 |        blackPenguin
75 |        sphere 1 0.95 1.0 0.8
76 |    matrixExcursion$do
77 |        whitePenguin
78 |        transl 0 0 0.17
79 |        sphere 1 0.8 0.9 0.7

```

The resulting image can be found in figure 6.11.

Torso and shoulders are scaled body parts:

```

                                     Tux.hs
80 |createTorso = matrixExcursion$do
81 |    scal 0.9 0.9 0.9
82 |    makeBody
83 |
84 |createShoulders = matrixExcursion$do
85 |    transl 0 0.4 0.05
86 |    leftArm

```

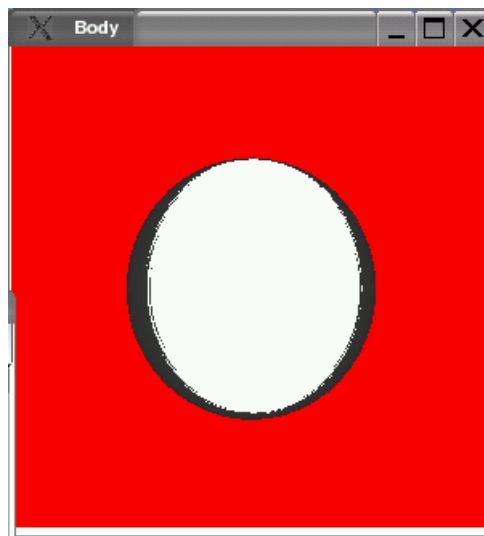



Figure 6.11: Basic part for a penguin torso.

```
87   rightArm
88   scal 0.72 0.72 0.72
89   makeBody
```

The resulting image for torso and shoulders can be found in figure 6.12.

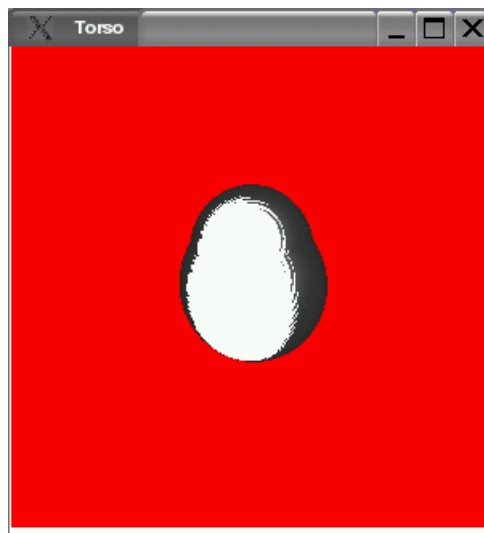


Figure 6.12: Penguin torso and shoulders.

```
90   createNeck = matrixExcursion$do
91   transl 0 0.9 0.07
```

Tux.hs

```

92   createHead
93   rotateY 90
94   blackPenguin
95   sphere 0.8 0.45 0.5 0.45
96   transl 0 (-0.08) 0.35
97   whitePenguin
98   sphere 0.66 0.8 0.9 0.7
99
100  createHead = matrixExcursion$do
101    transl 0 0.3 0.07
102  createBeak
103  createEyes
104  rotateY 90
105  blackPenguin
106  sphere 1 0.42 0.5 0.42
107
108  createBeak = do
109    matrixExcursion$do
110      transl 0 (-0.205) 0.3
111      rotateX 10
112      beakColour
113      sphere 0.8 0.23 0.12 0.4
114    matrixExcursion$do
115      beakColour
116      transl 0 (-0.23) 0.3
117      rotateX 10
118      sphere 0.66 0.21 0.17 0.38

```

Eyes

```

119  createEyes = matrixExcursion$do
120    leftEye
121    leftIris
122    rightEye
123    rightIris
124
125  leftEye = matrixExcursion$do
126    transl 0.13 (-0.03) 0.38
127    rotateX 18
128    rotateZ 5
129    rotateX 5
130    whitePenguin
131    sphere 0.66 0.1 0.13 0.03
132
133  rightEye = matrixExcursion$do
134    transl (-0.13) (-0.03) 0.38
135    rotateX (-18)
136    rotateZ (-5)

```

Tux.hs

```

137 rotateX 5
138 whitePenguin
139 sphere 0.66 0.1 0.13 0.03
140
141 leftIris = matrixExcursion$do
142   transl 0.12 (-0.045) 0.4
143   rotateY 18
144   rotateZ 5
145   rotateX 5
146   irisColour
147   sphere 0.66 0.055 0.07 0.03
148
149 rightIris = matrixExcursion$do
150   transl (-0.12) (-0.045) 0.4
151   rotateY (-18)
152   rotateZ (-5)
153   rotateX 5
154   irisColour
155   sphere 0.66 0.055 0.07 0.03

```

Legs

```

156 leftArm = matrixExcursion$do
157   rotateY 180
158   transl (-0.56) 0.3 0
159   rotateZ 45
160   rotateX 90
161   leftForeArm
162   blackPenguin
163   sphere 0.66 0.34 0.1 0.2
164
165 rightArm = matrixExcursion$do
166   transl (-0.56) 0.3 0
167   rotateZ 45
168   rotateX(-90)
169   rightForeArm
170   blackPenguin
171   sphere 0.66 0.34 0.1 0.2
172
173 leftForeArm = matrixExcursion$do
174   transl (-0.23) 0 0
175   rotateZ 20
176   rotateX 90
177   leftHand
178   blackPenguin
179   sphere 0.66 0.3 0.07 0.15
180
181 rightForeArm = leftForeArm

```

Tux.hs

```
182
183 leftHand = matrixExcursion$do
184   transl (-0.24) 0 0
185   rotateZ 20
186   rotateX 90
187   blackPenguin
188   sphere 0.5 0.12 0.05 0.12
189
190 leftTigh = matrixExcursion$do
191   rotateY 180
192   transl (-0.28) (-0.8) 0
193   rotateY 110
194   leftHipBall
195   leftCalf
196
197   rotateY (-110)
198   transl 0 (-0.1) 0
199   beakColour
200   sphere 0.5 0.07 0.3 0.07
201
202 leftHipBall = matrixExcursion$do
203   blackPenguin
204   sphere 0.5 0.09 0.18 0.09
205
206 rightTigh = matrixExcursion$do
207   transl (-0.28) (-0.8) 0
208   rotateY (-110)
209   rightHipBall
210   rightCalf
211
212   transl 0 (-0.1) 0
213   beakColour
214   sphere 0.5 0.07 0.3 0.07
215
216 rightHipBall = matrixExcursion$do
217   blackPenguin
218   sphere 0.5 0.09 0.18 0.09
219
220 leftCalf = matrixExcursion$do
221   transl 0 (-0.21) 0
222   rotateY 90
223   leftFoot
224   beakColour
225   sphere 0.5 0.06 0.18 0.06
226
227 rightCalf = matrixExcursion$do
228   transl 0 (-0.21) 0
229   rightFoot
230   beakColour
```

```

231 | sphere 0.5 0.06 0.18 0.06

```

Feet

```

Tux.hs
232 | foot = matrixExcursion$do
233 |     scal 1.1 1.0 1.3
234 |     beakColour
235 |     footBase
236 |     toe1
237 |     toe2
238 |     toe3
239 |
240 | footBase = matrixExcursion$do
241 |     sphere 0.66 0.25 0.08 0.18
242 |
243 | toe1 = matrixExcursion$do
244 |     transl (-0.07) 0 0.1
245 |     rotateY 30
246 |     scal 0.27 0.07 0.11
247 |     createSphere 0.66
248 |
249 | toe2 = matrixExcursion$do
250 |     transl (-0.07) 0 (-0.1)
251 |     rotateY (-30)
252 |     sphere 0.66 0.27 0.07 0.11
253 |
254 | toe3 = matrixExcursion$do
255 |     transl (-0.08) 0 0
256 |     sphere 0.66 0.27 0.07 0.10
257 |
258 | leftFoot = matrixExcursion$do
259 |     transl 0 (-0.09) 0
260 |     rotateY (100)
261 |     foot
262 |
263 | rightFoot = matrixExcursion$do
264 |     transl 0 (-0.09) 0
265 |     rotateY 180
266 |     foot

```

The resulting image can be found in figure 6.13.

Tail

```

Tux.hs
267 | createTail = matrixExcursion$ do
268 |     transl 0 (-0.4) (-0.5)
269 |     rotateX (-60)

```

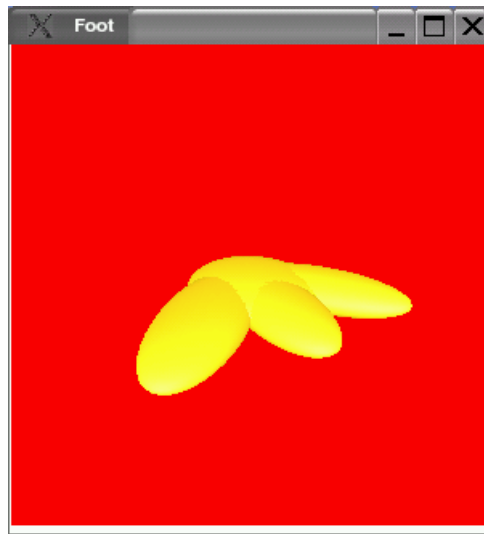


Figure 6.13: A penguin foot.

```
270   transl 0 0.15 0
271   blackPenguin
272   sphere 0.5 0.2 0.3 0.1
```

The complete penguin

We can use all the parts to define Tux.

```
----- Tux.hs -----
273 tux = matrixExcursion$do
274   scale 0.35 0.35 (0.35::GLfloat)
275   rotateY (-180)
276   rotateZ (-180)
277   createTorso
278   createShoulders
279   createNeck
280   leftTigh
281   rightTigh
282   createTail
```

The resulting window can be found in figure 6.14.

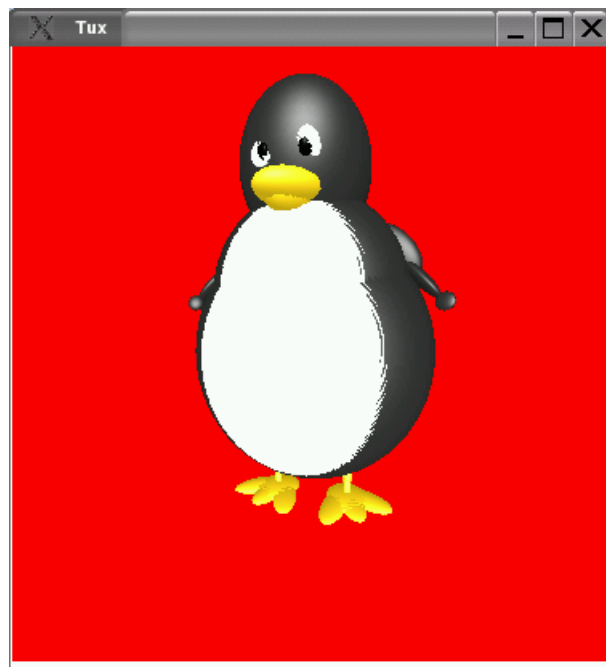


Figure 6.14: Tux the penguin..

Haskell Examples

APolygon, 1-3
ApproxCircle, 2-17
BackgroundColor, 2-5
Circle, 2-16, 2-17
Clear, 2-4
Coffee, 3-4
ColorCube, 6-9
Compose, 3-6, 3-7
Cube, 6-17
Double, 5-5
EvenMorePoints, 2-8
FillCircle, 2-18
ForgottenReset, 3-10
Get, 2-2
HelloWindow, 1-1
Hidden, 6-2
Idle, 5-4, 5-5
LightCube, 6-19
LineAttributes, 2-21
MorePoints, 2-7
MyTansformations, 3-9
NotHidden, 6-1
NotSmaller, 6-4
OrbitAroundCube, 6-10
OrbitPointOfView, 6-8
Ortho, 4-4
PointsForRendering, 2-9
PointSize, 2-21
PolyColor, 2-23
Pong, 5-6–5-10
Print, 1-5
RenderCube, 6-18
RenderLineLoop, 2-10
RenderLines, 2-10
RenderLineStrip, 2-11
RenderPoints, 2-10
RenderPolygon, 2-16
RenderQuads, 2-14
RenderQuadStrip, 2-15
RenderRubik, 6-14, 6-16
RenderRubikFace, 3-13
RenderStar, 2-25
RenderTriangleFan, 2-14
RenderTriangles, 2-12
RenderTriangleStrip, 2-12
Reshape1, 4-1
Reshape2, 4-3
Ring, 2-19, 2-20
RubikFace, 3-11, 3-12
RubikLogic, 6-12, 6-14
RubiksCube, 6-16
Set, 2-1
SetGet, 2-3
SinglePoints, 2-6
Smaller, 6-6
SomePoints, 1-2, 1-3
SomeRings, 3-1, 3-2
SomeSquares, 3-3, 3-4
Squares, 3-2, 3-3
Star, 2-24, 2-25
State, 5-3
StateUtil, 5-1
Tea, 2-26
TestCircle, 2-17
TestRing, 2-20
TestShear, 3-9
Tux, 6-20–6-25, 6-27, 28
Viewport, 4-3

List of Figures

1.1	A simple polygon.	1-4
2.1	Lines between points.	2-11
2.2	A loop of lines.	2-11
2.3	A strip in terms of lines.	2-12
2.4	Triangles.	2-13
2.5	A triangle strip.	2-13
2.6	A triangle strip.	2-14
2.7	Quads.	2-15
2.8	QuadStrips.	2-16
2.9	10 points on a circle.	2-18
2.10	Rendering a full circle.	2-19
2.11	A filled circle.	2-19
2.12	A simple ring shape.	2-21
2.13	Points of a large size.	2-22
2.14	Thick stippled lines.	2-23
2.15	A triangle with different vertex colors	2-24
2.16	RenderStar	2-26
2.17	A tea pot.	2-27
3.1	Rings translated to different positions.	3-3
3.2	Squares translated and rotated.	3-5
3.3	A coffee pot.	3-6
3.4	Different compositions of translation and rotation.	3-8
3.5	Applying shear to some shapes.	3-10
3.6	a Side of Rubik's Cube with further transformations applied to it.	3-14
4.1	Enlarging a window with the empty reshape function.	4-2

4.2	A Viewport smaller than the window.	4-4
4.3	Two windows with different projection.	4-6
5.1	Pong in action.	5-6
6.1	Third dimension not correctly taken into account.	6-2
6.2	Third dimension correctly taken into account by use of depth function.	6-3
6.3	Two squares in orthographic projection.	6-5
6.4	Perspective projection with frustum.	6-5
6.5	Two squares in perspective projection.	6-7
6.6	Perspective projection.	6-7
6.7	A view of the colored cube,	6-11
6.8	Rubik's Cube in action.	6-11
6.9	An unlit cube.	6-18
6.10	A lit cube.	6-20
6.11	Basic part for a penguin torso.	6-23
6.12	Penguin torso and shoulders.	6-23
6.13	A penguin foot.	28
6.14	Tux the penguin.. . . .	29

Bibliography

- [CFH⁺98] Norman Chin, Chris Frazier, Paul Ho, Zicheng Lui, and Kevin P. Smith. The OpenGL Graphics System Utility Library.
<ftp://ftp.sgi.com/opengl/doc/opengl1.2/glu1.3.ps>, 1998.
- [Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries The Revised Report*. Cambridge University Press, Cambridge, 4 2003.
- [Kil96] Mark J. Kilgard. The OpenGL Utility Toolkit (GLUT).
<http://www.opengl.org/developers/documentation/glut/glut-3.spec.ps>, 1996.
- [P JW93] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, 1993.
- [Wad90] P. Wadler. Comprehending monads. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM.
- [WBN⁺97] Mason Woo, OpenGL Architecture Review Board, Jackieneider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide 3rd Edition*. Addison-Wesley, Reading, MA, 1997.