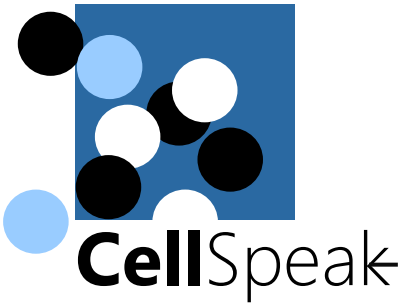


CellSpeak

Graphics Package

July 2017

Cells run in parallel,
On all cores,
On interconnected systems,
As one program

Table of Contents

1 Introduction

1.1 Packages and modules

A project in CellSpeak compiles to a module, and modules can be combined together in packages, i.e. a package can contain one or several modules. It is practical to combine related modules into a single package, or even to put a complete application in a single package, but the 'unit of dependency' in a CellSpeak program, is the module. Sometimes the terms *package* and *module* are used interchangeably, where in most cases the term *module* should have been used. Most of the time this does not lead to confusion.

Note that modules are identified by their *group* name in the source code, but not every group is a module – a module can indeed contain many groups.

1.2 Strong and weak links

This manual describes the packages that are available for development in CellSpeak.

Packages contain modules and modules contain types, functions and designs that are useful for many other projects.

Modules that share types with other modules are said to have a strong link, meaning that if the types in these modules change, then also the modules that use these types have to be recompiled.

Modules that only expose designs and interfaces to be used by other modules for instantiation and exchange of messages, are said to have weak links. When these packages change, the packages that use them do not have to be recompiled.

Of course, if the interface of a design – the messages it can receive and send – changes, then the module that wants make use of these changes will have to be modified and recompiled. Additions or changes to the internal workings of weakly linked modules, have no effect on the modules that use these modules.

It is clear that it is preferable to use in a project as much as possible weakly linked modules. Some modules however mostly contain types and functions, such as the *math* or *string* library. These packages are ‘standard’ packages and not expected to change often so they can be safely used by other packages without worrying too much about the dependency.

Often however packages will need to export some type definitions and design definitions in order to make their full functionality available. In that case it can be beneficial to separate the module in two or more modules: one module for the type definitions and one or more modules for the designs and interfaces. All these modules can of course conveniently be combined into one package.

1.3 Format of the description

In the following chapters we describe the messages that can be received by the interfaces supported by the different cell types and the replies that these cells can give to these messages:

```
design NameOfTheDesign( parameters ) is

     interface NameOfTheInterface is

          on MessageA( parameters ) do

             <- ReplyA1( parameters )
             <- ReplyA2( parameters )

          on MessageB( parameters ) do

             <- ReplyB1( parameters )
             <- ReplyB2( parameters )

          etc.
```

The parameters are given as a combination of type and a name for the parameter. As we have seen, the name of the parameter has no significance in the selection of a message handler and is added here just for clarity. The developer can choose whatever name he wants for a message parameter.

1.4 Overview

The *Graphics Package* contains designs, types and functions that are commonly used in graphics applications.

The package is built on top of well-known graphics libraries such as OpenGL or DirectX. The package offers designs that are typically required in graphics applications, like a camera, lights, geometry for objects etc, that offer a message interface to an application. In this way, the lower-level details of a graphics package are handled in the design and the designer can concentrate on the interaction and specifics of the application.

The designs in the package can be used as is, or can be used to derive more specific or elaborate designs more suited to the application at hand.

As with all packages in CellSpeak, the lower level details remain fully accessible to the designer, e.g. to create a new shader, and can be modified at the level of the CellSpeak source or at the level of the underlying C++ library.

The message based interfaces of CellSpeak are a natural fit for graphics application and also allow to integrate 3D graphics with other applications in a straightforward way. Also the high performance of the CellSpeak messaging framework make well suited for graphics applications.

Apart from designs, the package also contains types and functions for commonly used entities in graphics packages, like materials, geometry and the like.

Because the underlying C++ library of the graphics package is based on existing packages that have a different API, the Graphics Package to use is different according to the underlying library. However care is taken to make sure that message interface for working with the designs is the same irrespective of the package. For example a camera design has the same interface to define and position it, whereas the internals of the camera design to accomplish this will in general be different according to the underlying package.

It is however not always possible to make the interface completely independent of the underlying graphics package. Where there is a difference based on the graphics package, this is clearly indicated in the documentation.

The graphics package documentation is organized as follows:

- Geometry
- Material and color
- Shape
- Camera
- Light

Geometry, material and color are mainly objects and methods that are used as types in the messages that are exchanged between cells, whereas cameras, lights and shapes are about the designs and interfaces used to build a graphics application.

Before starting with the discussion of these graphics components, we have a look at how the underlying graphics libraries are integrated.

2 Graphics libraries

The Graphics package supports three base graphics libraries:

- DirectX 11
- DirectX 12
- OpenGL
- Vulkan

Each of these packages consists of a number of libraries that implement the functionality of the package. It is perfectly possible to use the objects, methods and functions that are exposed in these packages directly in CellSpeak through their ABI (Application Binary Interface), but it simplifies integration to create some C++ types and objects that are a better fit to the CellSpeak types used in the library.

For example to make a camera type, it makes more sense to do part of the initialization in the C++ library (setting up the buffers, shaders etc), because many of the preparations that have to be done are fairly low-level and in this way we can limit the number of definitions that are shared between the DirectX or other Graphics libraries and the CellSpeak Graphics libraries.

2.1 DirectX11

If you want to use DirectX11 as the basis for your graphics application, then you have to include the following bytecode packages in your project file:

```
-- The package files for this project - also load the symbols
const byte[] FullPackage[] = [
     "..\\Packages\\Platform\\Platform.celbyc",
     "..\\Packages\\D3D11Graphics\\D3D11Graphics.celbyc"
]
```

The graphics package uses the platform package, so that package has to be included as well. We import the packages as full packages, because we also want to use the types defined in these packages.

The types and classes that are used to exchange data are limited:

```
--------------------------------------------------------------------------------
-- 3D groups
--------------------------------------------------------------------------------
use Math, Color

group D3D11 is lib Direct3D11Lib

-- type mapping from the lib to CellSpeak types
lib type int, float, double, void
lib type cs_bar is byte[]
lib type "Vector3<float>" is xyz
lib type "Matrix4x4<float>" is matrix4
lib type "Matrix3x3<float>" is matrix3
lib type colour is Color.rgba
lib type MaterialType is Material.rgba

-- lib classes - have methods but are otherwise unspecified
lib class CameraClass
lib class MeshClass

-- The general structure of a mesh consisting of vertex and the normal
type PointType is record
     xyz Vertex
     xyz Normal
end

-- The basic meshtype
type MeshType is record
     MeshClass      cppMesh -- the class used by DirectX11
     PointType      Point[] -- co-ordinate and a normal data per vertex
     word16         Index[] -- vertex indices
end

-- The CellSpeak MeshType is equivalent with the MeshStruct of the D3D11 library
lib type MeshStruct is MeshType
```

First there is a mapping between the types in the library and the types in CellSpeak. Then two classes are listed that are made accessible from within CellSpeak code: the CameraClass and the MeshClass.

The MeshClass is the class that holds the geometry in a format that is usable by DirectX11. MeshType is the data structure that is used in CellSpeak to hold the geometry. No data is copied between MeshClass and MeshType, only pointers to the arrays of vertices and normals.

The internals of the type MeshClass are irrelevant in CellSpeak except for a limited number of methods such as *Transfer*, that fills in the structure based on the Points and Indices in a the record *MeshType*, or *Draw* that draws the mesh on the screen.

The CameraClass is the class that holds the bulk of the DirectX11 specific data and methods. The CameraClass holds the viewport, the window, the transformation matrices and so forth. It is also the CameraClass that is used to set up DirectX11 at the start to allocate buffers and so forth.

The CameraClass is encapsulated in CellSpeak in a design of the type Camera, to give it a message based interface to the other elements of a graphics application. For a limited number of messages, the CameraClass is used as a parameter, most notably the *Draw* message, to use it in the *Draw* method of the MeshClass.

What typically happens in an application is that a cell – for example an object made of one or several meshes – will receive a *Draw* message from the camera for every frame and as a result will call the appropriate methods for each of its meshes to draw itself. Example:

```
-- Draw is a request to draw - the parameter is the C++ CameraClass
on Draw(CameraClass D3D11Camera) do

     -- set the worldmatrix for this mesh
     D3D11Camera.SetWorldMatrix(&WorldMatrix)

     -- Set the material for the mesh
     D3D11Camera.SetMaterial( &MatSpec )

     -- And draw the mesh
     cppMesh.Draw(D3D11Camera)

end
```

3 Geometry

Geometry is based on the CellSpeak vector and matrix types: xyz, xyzw, matrix3 and matrix4.

Geometry is a collection of vertices and normals organized in some way. Geometry can be defined programmatically or it can be created and sculpted with specialized 3D programs. The former is mostly used for simple or mathematical forms while the latter is used for real-life complex forms.

Whatever the origin of the geometry however, the fundamental organization of it does not change. Real-life complex geometry is usually made available via files that have a specific format (e.g. the FBX format) that contains more information than just the geometry, so we will discuss this in a separate chapter and focus in this chapter on programmatically defined geometry.

The basis of the of the geometry is the *MeshType*. Derived from that we have the *ClosedMeshType* and the *OpenMeshType.* These two mesh types do not differ in structure, but they only differ in the way they are being constructed.

The *OpenMeshType* and *ClosedMeshType* are then used as the basis for a number of geometrical mesh types such as cubes, spheres, cylinders etc, but many more can be derived from these two types.

3.1 MeshType

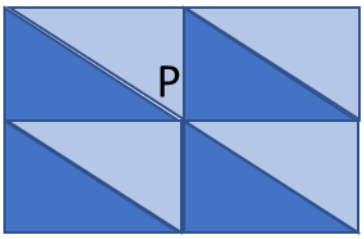
The MeshType is defined as follows:

```
-- The basic meshtype
type MeshType is record
     MeshClass      cppMesh -- the class used by DirectX11
     PointType      Point[] -- co-ordinate and a normal data per vertex
     word16         Index[] -- vertex indices
end
```

It only contains three fields and no methods of its own.

As we have seen, the MeshClass is the DirectX representation of the mesh.

The *Point* array is an array of vertices and the normal in each of these vertices. A triangular mesh consists of a series of connected triangles, so many of the vertices and their normals are going to be reused for several triangles:



In the figure above, the point P is used in six triangles. Instead of repeating the vertex and normal information, six floating point numbers, for each of these triangles, the triangles just use an index, a single integer, into the Point array. So if you have a surface defined by a list of points, then the index array contains a sequence of triplet indices, where each triplet defines a triangle for that surface.

Constructing a mesh type then consists of three steps:

- Making a list of vertices for the mesh. The number of vertices will depend for example on the level of detal you want to have.
- Determining the normal on the surface in each of the vertices. Note that the normal is used to calculate the effect of light falling on the surface.
- Filling in the index array to specify the triangles that make up the surface.

From the MeshType above we derive two new meshtypes: *OpenMeshType* and *ClosedMeshType*. Both types are identical to MeshType with the exception of one added method: *MakeIndices.*

This function fills in the *Index* array based on the *Point* array. For the *OpenMeshType* the points are supposed to represent an open table-cloth type of surface – whereas for the *ClosedMeshType* the points represent a closed surface, which has a top and a bottom cap.

Filling in the *Index* array requires a bit of index gymnastics, but the general principles are not difficult and those interested in the details can inspect the code in the source files.

3.2 Cubes, Cones and Cylinders

Based on the two mesh types in the previous section, we define a number of geometric meshes:

```
-- A mesh record for a cube
type Cube is MeshType with

     -- the method to construct the Point and Index array
     function Build is
end

-- A closed cone only has a closing cap at the bottom.
type ClosedCone is MeshType with

     -- this function builds a straight cone
     function Build(int Slices,float Radius)

end

-- The mesh record for a cylinder is based on the OpenMeshType defined above
type OpenCylinder is OpenMeshType with

     -- This is a build function for a straight cylinder
     function Build(int Slices, int Stacks)

     -- This is a build function for a cylinder made from a rotated curve
     function Build(xy[] Curve, int Slices)

end -- record

-- a closed cylinder is like an open but with a cap at the top and the bottom
type ClosedCylinder is ClosedMeshType with

     -- The build function for the straight cylinder
     function Build(int Slices, int Stacks)
end -- record


-- A closed sphere with two poles
type PolarSphere is ClosedMeshType with

     function Build(int Slices, int Stacks)

end

-- A sphere that is open at both ends
type OpenSphere is OpenMeshType with

     -- A sphere with an opening at both poles of OpenAngle radians
     function Build(int Slices, int Stacks, float OpenAngle)

end
```

Each mesh has is derived either from the *MeshType* directly or from the *OpenMeshType* or *ClosedMeshType* and has one or several additional functions *Build.* The *Build* function will first calculate the points and normals of the points on the meshes and then call *MakIndices* to fill in the *Index* array of the structure.

The *Build* function typically has two parameters, *slices* and *stacks*, which determine the horizontal and vertical number of patches for the surface. But is also possible, as for the generalized cylinder, to define a *Build* function that uses a curve to build a surface by rotating that curve around an axis.

It is clear that by using the same approach many more geometrical forms can be defined with different types of construction functions.

4 Color and Material

Apart from geometry, an object needs to have a color or material to be able to render it on the screen.

Different Graphics packages often use different representations of color. Therefore, color in CellSpeak also has several possible representations.

Material is defined based on color and consists of color values for the different types of lighting that can strike an object.

Color and material are a big subject in computer graphics and it is clear that more sophisticated approaches can be built based on these primitive types.

4.1 Color

As already mentioned, color can be represented in different ways. The first representations uses four components for each color : R, G, B and A for the red, green, blue and alpha channel component of the color. Each component is given one byte and therefore the color value fits into a 32 bit word.

Because some packages start with the alpha channels also that variant is defined.

A second representation uses the same components to specify a color, but this time each component is specified by a floating point value, for better precision.

Both color types are derived from the standard vector types in CellSpeak, meaning that manipulating color values is efficient and have a number of predefined operations.

```
group Color

-- The type argb_byte (Alpha, Red, Green, Blue) is derived from the built-in
-- byte vector type by renaming the components of that type
type argb_byte is vec4b with
     rename c1 to a
     rename c2 to r
     rename c3 to g
     rename c4 to b
end

-- Some systems might require the rgba order - so we also define that type
type rgba_byte is vec4b with
     rename c1 to r
     rename c2 to g
     rename c3 to b
     rename c4 to a
end

-- iso bytes als floats can be used in a color vector
type argb_float is vec4f with
     rename c1 to a
     rename c2 to r
     rename c3 to g
     rename c4 to b
end

-- ..again the order might be different
type rgba_float is vec4f with

     rename c1 to r
     rename c2 to g
     rename c3 to b
     rename c4 to a
end

-- we rename the most used version (float) to a shorter name..
type argb is argb_float
type rgba is rgba_float
```

```
-- Some color constants
const rgba Black    = [0,0,0,1]
const rgba Red      = [1,0,0,1]
const rgba Green    = [0,1,0,1]
const rgba Blue     = [0,0,1,1]
const rgba Yellow   = [1,1,0,1]
const rgba Fuschia  = [1,0,1,1]
const rgba Aqua     = [0,1,1,1]
const rgba White    = [1,1,1,1]
const rgba Orange   = [1,0.65,0,1]

-- when we add two colors, the alfa channel remains 1.0
operator + (rgba a, rgba b) out rgba is
     return <rgba>[a.r + b.r, a.g + b.g, a.b + b.b, 1.0 ]
end

-- multiplying by a factor f - the alpha channel remains unchanged
operator * (rgba c, float f) out rgba is
     return [c.r*f,c.g*f,c.b*f, c.a]
end

operator * (float f, rgba c) out rgba is
     return [c.r*f,c.g*f,c.b*f,c.a]
end

-- multiplying two colors, multiplies them component by component
operator * (rgba a, rgba b) out rgba is
     return [a.r * b.r, a.g * b.g, a.b * b.b, a.a * b.a]
end

-- Sometimes we just use a random color
function random out rgba is
     return <rgba>[ randf(),randf(),randf(), 1.0 ]
end
```

4.2 Material

The material consists of a color value for each of the type of light calculations that are typically used by Graphics libraries:

```
group Material

-- For materials we define 4 colors used by shaders
type rgba is record

     Color.rgba     Ambient
     Color.rgba     Diffuse
     Color.rgba     Specular
     Color.rgba     Reflect

     -- a function to set all colors to the same value
     function Set(Color.rgba Color) is

end
```

5 Shapes

Now that we have the geometry and the material defined, we can start to define the first type of designs that we want to use in applications: shapes.

Apart from a geometry and a color, shapes also have a message interface – they are based on designs – and therefore can have a behavior.

Shapes are the building blocks of a scene, and basically a limitless variety of shapes can be built based on geometry and behavior – varying from simple static forms to complex highly dynamic shapes.

In the following paragraphs we show how some basic shapes can be constructed and what are typically the type of interfaces a shape should support.

5.1 Shared geometry shape

The first shape is an efficient shape in the sense that it does no keep a copy of its own geometry, but rather shares his geometry with all similar shapes.

Apart from the advantage of using very little memory even if large numbers of the shape are used in an application, there are of course some limitations: the geometry of the shape is the same for all its members, but because each shape has its own world matrix, the size orientation and position of each shape is unique. Each shape also has its own material.

Because the geometry for the shape is only created once, the geometry is available in each shape as a pointer to the C structure that contains that geometry.

Using the building blocks we have defined before, the implementation of the *Shape.SharedMesh* is surprisingly simple. As an example we reproduce here the actual code for that design:

```
group Shape

-- The basic shape that all others are derived from
design SharedMesh(MeshClass cppMesh) is

     -- A shape is defined by default in position (0,0,0)
     xyz Position = [0,0,0]

     -- The worldmatrix transforms the shape to its position in the 'world'
     matrix4 WorldMatrix = I4X4

     -- A shape is made of some material – default we set it to white
     Material.rgba MatSpec = Material.White

     -- we keep the shared mesh class that was passed as a parameter
     keep cppMesh

     -- Draw is a request to draw - the parameter is the C++ CameraClass
     on Draw(CameraClass D3D11Camera) do

          -- set the worldmatrix for this mesh
          D3D11Camera.SetWorldMatrix(&WorldMatrix)

          -- Set the material for the mesh
          D3D11Camera.SetMaterial( &MatSpec )

          -- And draw the mesh
          cppMesh.Draw(D3D11Camera)

     end

     -- a message to set the world matrix of the shape
     on SetWorldMatrix( matrix4 WorldMatrix ) do
          keep WorldMatrix
     end

     -- a message to move the shape to a certain position
     on MoveTo(xyz Position) do
```

```
          -- we keep the new position ..
          keep Position

          -- and adjust the Worldmatrix for this translation
          WorldMatrix.c14 = Position.x
          WorldMatrix.c24 = Position.y
          WorldMatrix.c34 = Position.z
     end

     -- If the material is changed
     on SetMaterial(Material.rgba MatSpec) do
          keep MatSpec
     end

end
```

This design is simple – it has no constructor or destructor and just a few messages, but it is fully functional. The design can also be used to derive more complex designs from, for example by adding extra interfaces.

Each shape can be given its own initial position and size by setting the world matrix with the *SetWorldMatrix* message, and it can equally be given its own material.

The *MoveTo* message is typically used to add movement to the shape.

An important message is the *Draw* message. As we will see, it is the camera design that issues the draw message, e.g. every 40 msec, but it is the shape that draws itself using its state and the CameraClass passed as a parameter.

6 Camera

The camera is an important design in the Graphics Package. The camera keeps track of the low-level settings of the graphics, the drawing mode, the window it is using, the viewport etc.

The camera reacts to a number of messages that typically come from the mouse and keyboard interface and that are used to for example to pan and move the camera, or to change the drawing mode from wireframe to shaded. The camera itself also generates messages, the most typical of which is the *Draw* message, that it will send to the objects in a scene.

As in real life, a Camera in a graphics package can be simple or complicated with many settings that allow to tweek its performance and characteristics. Therefore the *Camera* is a group of objects from which the most appropriate camera can be selected for a given application.

6.1 Basic Camera

The basic camera contains the elements that are common to all cameras.

```
group Camera

-- The basic camera model contains the D3D11 camera (see the D3D11 lib) and the
-- fundamental matrices used by the camera to render its view of a scene.
design Basic is

     CanvasWindowClass    CameraWindow -- a camera displays its image in a window
     RectangleType        Rectangle    -- the rectangle of the window
     CameraClass          D3DCamera    -- the D3D11 camera device (a C++ class)

     matrix4 ViewMatrix                -- The view matrix
     matrix4 ProjectionMatrix          -- The projection matrix
     matrix4 CameraMatrix              -- The matrix = W x P x V

     -- Cameramodes
     type CameraModes is [SingleFrame, Animation]
     var Mode = CameraModes.SingleFrame
     bool Solid

     cell Scene = null                 -- The top level cell of the scene
     cell Timer = null                 -- The camera will need a frame timer

     -- The camera can do solid modelling..
     on SetSolid do
          D3DCamera.SetSolid()
     end

     -- or wireframe modelling
     on SetWireFrame do
          D3DCamera.SetWireFrame()
     end

end
```

The basic camera does not send or receive messages and as such is of no direct use, but from this camera we derive other camera types.

6.2 CameraFromTo

6.2.1 Overview

```
design FromTo(xyz From, xyz To, xyz Up, float FarZ, float NearZ, float FieldOfView
) like Basic is

     -- This message sets the window to be used by the camera
     on SetWindow(CanvasWindowClass CameraWindow)

     -- a message that is sent to the camera when the window was resized
     on WindowWasResized(CanvasWindowClass CameraWindow)

     -- a message sent to the camera when the window was moved
     on WindowWasMoved(CanvasWindowClass CameraWindow)

     -- A message to set the scene that the camera has to draw
     on SetScene(cell Scene) do

     -- message that reports mouse moves
     on MouseMove(point FromPoint, point ToPoint, word Keys)

     -- message for mousewheel clicks
     on MouseWheelClicks(int Clicks)

     -- a message sent when a key was pressed in the camera window
     on KeyPressed( word Key )

     -- a message to get the camera going
     on Roll

     -- to be executed on every frametick
     on FrameTick( word Time )

     -- a message to add a light to the scene
     on SetLight(cell NewLight)

     -- Reply from a DirectionalLight
     on DirectionalLight(Color.rgba Ambient,Color.rgba Diffuse,Color.rgba
                         Specular,xyz Direction)

     -- Reply from a PointLight
     on PointLight(Color.rgba Ambient,Color.rgba Diffuse,Color.rgba Specular,xyz
                   Position,float Range, float Attenuation)

     -- Reply from a SpotLight
     on SpotLight(  Color.rgba Ambient,Color.rgba Diffuse,Color.rgba Specular,xyz
            Position,float Range,xyz Direction,float Spot,float Attenuation)


end -- camera from to
```

This camera is also used to derive a camera for which a number of default values have been filled in. The only remaining parameters are the position of the camera and the point in space it is pointed at:

```
-- a derived camera with some default settings
design FromTo(xyz From, xyz To) like
     FromTo( From, To, <xyz>[0,1,0], 1000.0, 0.1, <float>pi/2) is end
```

6.2.2 Constructor

FromTo(*see parameters*)	
description	Creates a cell for the camera
parameters	• xyz From • xyz To • float Farz • float NearZ • float FieldOfView
replies	
remarks	

When camera is created, the underlying graphics framework is also initialized if not yet done so.

6.2.3 → SetWindow

SetWindow(CanvasWindowClass CameraWindow)	
description	Sets the window to be used by the camera for drawing.
parameters	• CanvasWindowClass CameraWindow
replies	
remarks	