GLScene Guide

v.2020.07.07

GLS

Contents

Disclaimer

This guide is intended to quick start for using GLScene as official documentation. It’s also strongly recommend using source code as a primary source of information in comments directly in source code. Especially when the source code is being developed constantly and is changing all the time. Another great source of knowledge are GLScene\Demos. However GLScene is enough complex that is rather difficult for anyone who never used it before to start learning.

If you are an experienced Delphi or C++Builder user you will probably find this document very simple and we would be happy for any feedback from you. There are sections without description and these parts are marked. If you are willing to contribute to this book, please, feel free to send your chapter in discussion forum and we will add it to the text together with your credit. Any corrections of the current text are also welcome.

What is GLScene ?

Reading this document you have probably already seen some GLScene applications or even worked with GLScene itself. Simply said GLScene is Delphi/C++Builder library for OpenGL graphics. OpenGL is graphics API originally developed by Silicone Graphics. OpenGL is a standard and it should be installed on almost any PC nowadays. GLScene makes developing OpenGL applications very easy and simple. GLScene applications made with Delphi are primarily targeted for Windows platform, there is clone of GLScene for Lazarus so porting to other platforms could be possible.

GLScene was founded by Mike Lischke and from the very beginning it was developed as an open source library for programmer's community. Later on it was taken over by Eric Grange and is steadily growing ever since. There is a number of GLScene's administrators now. GLScene is hosted by SourceForge at www.sourceforge.net/projects/GLScene and by GitHub at https://github.com/glscene. There is countless number of projects made with GLScene ranging from simple game demos to complex scientific applications you may find at pages https://sourceforge.net/p/glscene/code/HEAD/tree/branches/ and https://github.com/GLScene/ExCBuilderGLS.

Installing GLScene

There are two ways how to obtain GLScene source code. Which one will you choose depends on your intentions with this library. If you just want to try it out download the snapshot version. Snapshot is an archive file containing all source code files. Snapshot is available from www.glscene.org. It is packed with 7zip format (www.7-zip.org). Just unpack it wherever you want your GLScene to reside.

Another way how to download GLScene is using SVN or GIT. Svn is an Internet protocol system used by SourceForge for on line accessing source code repositories by developers. Choose this method if you want to have the latest version of GLScene and check frequently for changes and updates. You will need recommended TortoiseSVN client in order to download or TortoiseGIT client to clone it on your computer.

All SVN/GIT clients have one thing in common. It is called SVN *root*. It is a string, in fact an address that is used to access web.

If you have TortoiseSVN installed right click in directory where you want GLScene to be downloaded and select SVN *Checkout*. Enter the above SVN root. Once you have all GLScene files on your hard drive you can compile the packages in Delphi. Choose corresponding directory to your version. There are several packages in GLScene.

1. GLScene_RT/DT.dpk – core package containing GLScene itself.
2. GLScene_PHYSICS_RT/DT.dpk – ODE is physics plug-in for GLScene.
3. GLScene_PARALLEL_RT/DT.dpk

4. GLScene_SOUNDS_RT/DT.dpk

5. GLScene_Cg_RT/DT.dpk – Cg shaders.

You may have to manually add GLScene directories in compiler search path. After successful compilation there will be four new tabs with components. By default they will be placed in the rightmost position next to other component tabs. You will have to scroll to the right in order to see them or rearrange the tabs. Now you are ready to use GLScene!

How does it work?

You are probably used to Delphi/C++Builder VCL design time functions. Designing form is a straightforward procedure. Visual components that you put on the form can be right visible. With GLScene things are not that simple. Some components are visible at design time some are not. This chapter will explain the philosophy of GLScene and the way GLScene is organized.

According to Delphi naming conventions every GLScene class will start with TGL... prefix.

First of all you should keep in mind that everything in GLScene has a strict hierarchy. Every object belongs to other object which belongs to another object. This is called parent <-> child relationship. Every object can have unlimited number of children but only one parent. The top of the object tree is *TGLScene*. The analogy to this would be a *TButton* placed on a *TPanel* which is placed on *TForm*. There is one interesting issue in the hierarchy. An object always has a parent and a owner. Parent and owner are not the same. Owner must always be *TGLScene* because it is registered there. Parent however can be another object or TGLScene.Objects class.

There are basically two kinds of objects in GLScene : **components** and **scene objects**. Components are organized in four component tabs and can be placed on the form and accessed only in object inspector. They can be considered 'servicing components' for the second type of objects. Scene objects can be added, edited and deleted in the scene editor. Scene objects actually represent the content that is going to be rendered in the scene. Scene objects are also displayed in object inspector but object *inspector* lacks the functionality of scene editor. This is my own classification and has nothing to do with Delphi’s interpretation of words ‘object’ and ‘component’.

Scene editor is the heart of GLScene. To open scene editor *TGLScene* has to be placed on the form. Double click on it in object inspector. New window pops up. There are editing buttons in the top row and a tree view box below. You can add, select, move and delete scene object in the tree view box. You can also display each object's effects and behaviors here.

‘Hello world !’ example

Many programming guides contain the obligatory ‘Hello world !’ example. It is usually simple application where user presses button and message window saying ‘Hello world !’ appears. This is necessary for readers that can’t wait to start with something in practice. In my opinion the equivalent to this in 3D graphics programming is a spinning cube.

So here is step by step guide how to create spinning cube in GLScene:

1. Start new application in Delphi.
2. Double click on *GLScene* to add *GLScene1* – white cube on GLScene tab first from left.
3. Double click on *GLSceneViewer* to add *GLSceneViewer1* – white cube with camera on GLScene tab second from left.
4. Select *alClient* as *Align* property for *GLSceneViewer1*.
5. Double click on *GLCadencer* to add *GLCadencer1* – metronome on GLScene tab fifth from left.
6. Select *GLScene1* as *Scene* property of *GLCadencer1*.
7. Double click on *GLScene1* in object tree view to open GLScene editor.
8. Right click on *Cameras* in GLScene editor and click on *Add camera*.
9. Select *Camera1* and set *Position.X := 5; Position.Y := 5; Position.Z := 5;* to move the camera back so it can overlook the scene.
10. Right click on *Scene objects* in GLScene editor and select *Add object* then *LightSource*.
11. Select *GLLightSource1* and set *Position.Z := 10* to move the light up so it can shine on the cube from above. 12.Right click on *Scene objects* in GLScene editor and select *Add object* then *Basic geometry* then *Cube*. 13.Select *Camera1* and set *GLCube1* as *Target* property to make the camera look at the cube. 14.Select *GLSceneViewer1* and select *GLCamera1* as *Camera* property. 15.Double click on *GLCube1* in object tree view to add progress event on the form. 16.Write *GLCube1.TurnAngle := GLCube1.TurnAngle + deltaTime * 100;* this will make the cube spin around Z axis. 17.Press F9 and see your first spinning cube!

One might say that a spinning cube is a trivial thing. It indeed is with GLScene but let’s reconsider what it takes to draw a spinning cube on modern hardware and operating system. First GLScene itself has good few thousands of lines of code. Not all of it is used to render a cube of course. Then we have OpenGL with all the possible drivers. And in the end there is the operating system on top of all that. So you can see there was a lot of work and effort done by other people so you can make your spinning cube.

Basic things you should know about 3D graphics

GLScene is a very powerful tool but you still need to know at least something about 3D computer graphics. Skip this chapter if you are familiar with this topic. There is a lot of theory about 3D graphics and here provided only some basic things specific to GLScene and OpenGL.

Coordinate system

A three dimensional world is described with three axes: X, Y, Z. They intersect in the origin point at position [0,0,0] with 90° angle. The orientation of these axes can be a bit confusing. Math conventions anticipate that the up and down axis is Y, left and right axis is X and the depth is described by Z. But OpenGL standard says that Z is up and down and Y is the depth. X axis remains the same. So you should remember that OpenGL coordinate system is different then for example the one other 3D editing applications use. You can display objects' axes by setting *ShowAxes:=true*. They will be represented by red, green and blue dotted lines both design and run time. To make things even more complicated some objects use the conventional coordinate system like *TGLGraph* and *TGLTerrainRenderer*.

In the end which direction is up will depend on the camera rotation. You can position your object in the scene in whichever orientation you like. By default a new object will be placed in the direction facing positive Y axis and top of the object pointing to positive Z axis.

It is important to understand the difference between global and local coordinate system. Every child object exists in its own local coordinate system. If parent changes it's position or rotation the child will move with the parent but it's local position will remain the same although it changed global position. This may look a bit complicated at the beginning but it is an important feature that allows keeping the scene hierarchy well organized.

Floating point numbers

OpenGL uses single precision system. What does it mean? It means that every decimal number is 8 bits long. Let's explain this more. Decimal numbers can have unlimited number of digits after the decimal separator. The π constant is a good example. Computer can calculate the π constant for as many digits as it's memory is capable of. But that is not very practical. So where do we set the limit? Everyone can understand the difference between 3.14 and 3.1415926535. Delphi has strong support for types. In Delphi we have **single**, **double** and **extended** types for decimal numbers. Single is 8 bit long so it means the program allocates 8 bits of memory for it. The number of digits is limited. The π constant would be 3.1415926 in single type format. But what happens if we add 1000 ? The result should be 3000.1415926. But there are too many digits in this decimal number for the single type format. What happens is that the last digits are trimmed and we get the right length : 3000.1415. This model is simplified because the whole math is happening on binary level but you get the picture.

Remember that single precision is not accurate enough when working with high numbers or when you need extra precision. If you are doing some scientific calculations you can use extended type to get more exact results. But if there is an OpenGL graphical output like a graph all extended numbers will be down sampled to single precision.

It is not recommended to use Delphi's native Math unit with GLScene because the functions in this unit are too slow for real time rendering. Avoid using Math unit if you can.

Vectors

Now when we know what floating point numbers are we can explain what they are used for.

In GLScene a lot of calculations is done with vectors. Vector is an universal type that can describe almost anything from object's position, rotation, speed even color. There are different types of vectors: Vector3f, Vector4f, AffineVector. But vector generally is an array of three singles [X,Y,Z]. Some vectors have the mysterious fourth member called W. W is used for describing rotations or alpha value of the color. Many GLScene functions are overloaded and you can supply most of the vector types as a parameter.

OpenGL internally does not operate on vectors but on matrices. Matrix is a two dimensional array of 4 x 4 singles. Matrix transformations take care about useful things like vector rotations, scaling etc. You do not really need to know about matrix transformations because GLScene has functions to do the job for you. These functions are in *VectorGeometry* unit. Detailed description of vector geometry functions is far beyond the scope of this book but there is a good article at www.flipcode.com about vector math.

Rendering

Every object in 3D graphics is composed of polygons. Even curved surfaces are made up of polygons only very small ones. Polygon is a triangular face delimited by three vertices. Vertex is a point in space. Each polygon has front and back sides defined by face normal. Normal is a vector pointing in front direction of the face. Another information stored by face is connectivity with other faces. Polygon knows which surrounding polygons are attached to which side of it. This is used for smoothing edges between two adjacent polygons. Polygon also stores information about how texture is stretched over its surface. Texture coordinates of every vertex are defined by two numbers in range from 0 to 1. These are called U and V coordinates and specify exact position of the vertex on texture.

When the polygon gets rendered possible surrounding lights are considered, the angle between face normal and direction to camera and lights is considered and the polygon is lit and textured according to shading model. Different shades of polygon along curved surface create illusion of plasticity.

To render whole object which is made up of polygons the object has to be first transformed in position, rotation and scale. Every rendered pixel is stored in Z buffer. Z buffer is memory allocated for all pixels on the screen together with Z depth – distance from the camera. Every time a new polygon is rendered the resulting pixels overwrite possible existing ones only if the Z depth is smaller than the one stored in Z buffer for that particular pixel. More understandably new polygon is rendered only if it is closer than other polygons.

OpenGL

This is a wide topic, too complicated for this book. You don't necessarily need to know much about OpenGL to use GLScene but it is certainly better to have some understanding of it.

Components

This chapter is divided in four parts describing four groups of GLScene components: **GLScene**, **GLScene PFX**, **GLScene utils** and **GLScene shaders**.

GLScene Panel

GLScene

The basic component. You can open GLScene editor by double clicking on it.

GLSceneViewer

The scene viewer represents a rectangular canvas where the scene gets rendered. It's size or width - length ration is not restricted. The bigger the viewer the slower the rendering will be. You must specify *Scene* and *Camera* properties. Change important rendering context through *Buffer* property. However leaving *Buffer* default properties will suffice in most cases.

GLFullScreenViewer

Same as TGLSceneViewer but switches to fullscreen mode. The resolution has to be readable for the graphics card and monitor. So it should be something like 800 x 600 or 1024 x 768.

GLMemoryViewer

Memory viewer is a virtual canvas. You can render on it, read it’s output from the memory. But you can not see any picture on the screen. It is used for example for rendering cube maps for real time reflections or more advanced shadow techniques.

GLMaterialLibrary

Material library is a storage component for materials. It has collection of materials and you can access individual materials by index or the material name.

At this point I should explain something about how materials are handled in GLScene. Each object where material is applicable has its *Material* property. You can edit materials directly in object inspector or double click on the ellipsis icon next to material and open material editor. Material editor has three tabs and a window with an example cube textured with current material. The three tabs are:

1. **Front properties** - edit material quality of object’s front faces. Diffuse color being the most important. It defines color of the object on direct light. Ambient color defines the color of object in dark spots. Specular color is color of high reflections and shininess defines how big those reflections are. Emission color makes the object glow. But for now just remember that to change object’s basic color use diffuse color. Other material qualities like reflection and shininess can be achieved more realistically with material shaders.

2. **Back properties** - same as front properties but for faces that would be normally invisible. The back faces become visible only if the material is transparent and if back face culling is off.
3. **Texture** - in order for the texture to become visible you have to set *disabled* property off. This property is by default on which means that the object is not using any texture. It is colored and shaded with settings described by front and back properties. If you want to apply a texture to the material uncheck *disabled* checkbox and load an image. Formats supported by GLScene are bmp, jpg and tga. Then you have to set texture modulation to *tmModulate* for realistic lightning. Remember that a light has to illuminate the object so you can see the material. Another important thing is that texture should have the size of power of two. It means 4,8,16,32,64,128,512… The length and width don't have to be the same. You can have a texture 32 x 512. If you use a texture with size other than power of two it will be displayed but it will also be much slower because GLScene will have to resize it.

At bottom of material editor there is material **blending mode** listbox selection. You can specify how the material will blend or overlay other materials. Opaque blending mode creates a solid surfaced object. Transparent blending will make it possible to see through the object. The object can be uniformly transparent or the transparency can be defined by texture. Additive blending combines the colors of the object with colors behind it.

Although each object can have its own material it is strongly advised to store the materials you use in material library. Especially if many objects share the same texture. If you have 100 cubes with the same texture and you load the texture to each cube it will occupy the memory of 100 textures. You can load this texture in material library once and let the 100 cubes refer to that texture. To do this set *MaterialMaterialLibrary* to the material library and *Material.LibMaterialName* to the name of the material you want to use. Warning ! Objects have *MaterialLibrary* property themselves, you have to *use MaterialMaterialLibrary !*

Material library has one handy function: *AddTextureMaterial* In this function you specify new material name and image file to be loaded. A new material is added to the material library with *Texture.Disabled:=false* and *TextureModulation:=tmModulate.*

GLCadencer

Most of the GLScene applications do real time rendering. The time element takes important part here. We need some kind of time manager to deal with it. It is not a simple thing at all. First of all we do not know how long will it take for the computer to render the scene. The camera may be looking at complex geometry with a lot of polygons or the user may turn the camera away and no polygons will be rendered. The program may run on an old hardware configuration or a top high-tech PC with the latest CPU and graphics accelerator. These are factors you don't know in advance and you must take them into account.

If you want to progress you scene over time add *TGLCadencer* on the form. It will take care of proper synchronizing and updating objects frame by frame. You must assign it to *TGLScene.Cadencer* property.

The process of rendering a frame results in event called *Progress* in GLScene. Every GLScene object has *onProgress* event where you can code all extra actions to be executed every time the scene is rendered. Double clicking on an object in object inspector adds *onProgress* event to the form.

The progress procedure passes one important variable: *deltaTime.* It is a time period in seconds that took since the last frame was rendered. The longer it is the more jerky the motion in the scene will be. Ideal frame rate is 30 frames per second. *DeltaTime* would be 0.033333. Any time

you want to do any calculations related to time you have to include this parameter in your equations. For example if you want to move a cube along X axes with speed of 10 units per second. The formula would be like this: *Position.X:=Position.X + 10 * deltaTime.*

Cadencer has *enabled* property. You ca simply turn cadencer on and off. When it is turned off the scene will freeze. There are also several modes cadencer can run in. *cmASAP* is the default mode and it will progress the scene whenever it can giving it the highest priority. *cmIdle* will progress scene only when other processes are finished and with *cmManual* the scene will only progress when you make it to do so. Another interesting feature is *minDeltaTime.* If you set value for *minDeltaTime* the scene will not progress until that time has passed even if all rendering is done. This can give some rest for the system. *maxDeltaTime* on the other hand will not allow the cadencer to run faster then specified.

GLGuiLayout

GUI stands for 'graphical user interface'. The purpose of GUI components is to create
2D window controllers well known from Delphi - forms, panels, labels, check boxes etc. In GUI manager you specify GUI layout which is a file with .layout extension created with GUI editor. This layout specifies regions on a bitmap which is used for the graphic output. This topic will be covered in more detail later in GUI objects chapter.

GLBitmapFont

You can display 2D text with *TGLBitmapFont* This component is used together with *TGLHUDText.* You have to load a bitmap that contains letters arranged in rows and columns. The letters must be on black background for correct transparency. *Ranges* property specifies which regions of the bitmap correspond to which letter. Each range has width and length in pixels of the letter together with starting and ending character in alphabet in ASCII standard. Bitmap font will create array of textures each representing one character. *TGLHUDText* can read these textures and display words. Disadvantage of *TGLBitmapFont* is that all the letters must have same width. 'I' is same wide as 'M'. The advantage is that you can make a bitmap you like with colored characters and different transparency.

GLWindowsBitmapFont

If you don't want to bother with making the bitmap you can use existing Windows font. You just select *Font* property and *TGLWindowsBitmapFont* will create all textures for *TGLHUDText.* With this component any true type fonts will be displayed correctly but you have to use some common fonts that will most likely be installed on the target computer.

GLStoredBitmapFont

This is the third and in my opinion the most useful font component in GLScene. You can distribute your font with the application saved in .glsf file. This file is created with very easily from any Windows font. This way you can use some more exotic fonts and you do not have to worry if they are installed on other computers or not. Both *TGLWindowsBitmapFont* and *TGLStoredFont* can't have characters with different colors. You can only change a color as a whole.

GLScriptLibrary

GLSoundLibrary

GLScene is not only great for graphics you can also add 3D sound to your applications.

Sound library is similarly like material library a storage room for sound tracks. Load different sound files in *Samples*. Wav and mp3 formats are supported. Beware that longer mp3 sounds like music may produce errors when playing through sound library. Sound library has to be used together with *TGLSMWaveOut, TGLSMBASS* or *TGLSMFMOD*. To play the sound add sound behavior to scene object, specify which sound to play by index or name and the sound will be emitted by the object in 3D space.

GLSMWaveOut

GLODEManager

ODE is 'Open Dynamics Engine'. It is physics simulation library. Its home page is www.ode.org . You can find documentation and help files there. ODE is distributed in single dll file. ode.dll must be in the same directory as your application executable file or in Windows system directory. Simply said any scene object with ODE behavior will naturally react to gravity, collide and bounce off other objects. You can create cars, rag doll effect simple machines. To make ODE object in the scene add ODE behavior to it. There are two kinds of ODE behaviors:

1. **Static** - objects are supposed to remain stable in the scene. They will not move whatever may hit them. Example would be ground floor or walls.
2. **Dynamic** - objects have their weight and can move around the scene interacting with other objects or joints. Example is a ball.

Apart from the kind of behavior you must select proper collider. It is a shape attached to the object that stands for the mass of the object. Collider can be simple cube, sphere, cylinder or can be extracted from mesh geometry or heightfield data. One object can have multiple colliders creating complex shape.

ODE manager runs the simulation through ode.dll and updates each object's position and rotation. You must call *Step* procedure every frame for the system to progress. This is usually done in cadencer's *onProgress* event.

GLODEJointList

If you want to use joints in ODE you must add them in the joint list. Each item describes the type of the joint, the two objects it attaches and other parameters like joint restrictions, angles, offset positions etc. For more information about joints go to www.ode.org .

GLSMBASS

BASS is a sound library. Again bass.dll must be either in application or Windows system directory. TGLBASS is sound manager that is used together with *TGLSoundLibrary* and object sound behavior to play sounds.

GLSMFMOD

Same as BASS only uses fmod.dll.

GLScene PFX

In general all components on this tab are managers for particle effects. PFX manager takes care about the way particles look like, how they move, how fast they are or how long they live. Some particles can even change appearance during their live. Each object can emit particles. You can add particle effects to *Effects* property. Each effect then has its own PFX manager and PFX renderer.

GLCustomPFXManager

GLPolygonPFXManager

Polygon particles are made up of circular array of triangular polygons always facing camera. Polygons use vertex colors. You can specify inner and outer color for the particle. Outer color is usually transparent. Each polygon is opaque in the center of the particle and gradually becomes transparent towards the edge.

GLPointLightPFXManager

GLCustomSpritePFXManager

GLPerlinPFXManager

Particles created with perlin PFX have randomly generated texture. Great for smoke effects.

GLFireFXManager

This is manager especially designed for fire effect. It is best to leave default settings for the most pleasing visual quality. Adjust only the particle size as you need.

GLThorFXManager

Create lightning or laser beams with this effect. This effect stretches from one point to another. A sparkling beam glows between.

GLScene Utils

GLAsyncTimer

This component is similar to *TTimer* you know from Delphi. Only it is more suitable for

GLScene. Time *Interval* in milliseconds triggers *onTimer* event. This event is independent of

cadencer progress event. Async timer is not recommended to replace the cadencer. It is used to execute actions in fixed longer time intervals rather than take care about rendering the scene.

GLStaticImposterBuilder

Imposters are clever way how to improve performance dramatically. The basic idea behind imposters is that before rendering a set of images looking at an object from all sides is rendered and stored in memory. If the object is to be rendered a sprite (2D image) is drawn instead of the mesh. The most appropriate image has to be selected according to the object's orientation to the camera. This technique is great for many same objects with high polygon count scattered around the scene. On the other hand it consumes a lot of memory.

The task of imposter builder is to generate all the images. You can set the size of the image and number of angles images are to be rendered from. This is done by adding coronas. Each corona has minimum and maximum angles as well as number of images in this range. For example if you make corona ranging from 0º to 15º with 24 images the imposter will cover spherical segments 15º wide and 15º high around 'equator' of the object. You must call *RequestImposterFor* function to generate the coronas.

Following section describes five HDS components. **H**eight **D**ata **S**ource components provide data for *TGLTerrainRenderer*. Terrain is used for rendering large outdoor sceneries. HDS components load terrain data from different formats and supply suitable data for terrain renderer.

GLBitmapHDS

This is the easiest format. Bitmap HDS loads grayscale image and converts the data into height field. The grayscale image must be 16 bit. Most paint programs use 8 bit grayscale palette. Make sure you convert the image to 16 bit otherwise the terrain will look terraced. White parts of the image represent the highest altitude and black parts represent zero altitude.

TGLCustomHDS

GLHeightTileFileHDS

The source for height field HDS is .htf file. These files are created in . This format is most suitable for really large terrains. Ground texture coordinates together with *TileSize* property are included in .htf file. *TileSize* of terrain renderer must be the same as *TileSize* in .htf file.

GLBumpmapHDS

GLPerlinHDS

GLCollisionManager

GLAnimationControler

Animation controller is used together with *TGLActor* object. Animation controller stores animations that are used by actor. Animations can be loaded from Quake md2 format or from Halflife smd format. Animations are accessed by index or name. They can loop, play once or

backwards.

GLJoystick

GLScreenSaver

If you add this component on the form the application will be compiled as Windows

GLAVIRecorder

You can stream the rendered sequence in video file.

GLTimeEventsMGR

GLVfsPAK

GLNavigator

GLUserInterface

GLFPSMovementManager

GLMaterialScripter

GLDCEManager

GLApplicationFileIO

GLScene Shaders

Shaders are material modifiers. Each material can have a shader. Shader alters the appearance of the material creating various visual effects.

GLTexCombineShader

GLMultiMaterialShader

GLUserShader

GLOutLineShader

GLHiddenLineShader

GLCelShader

GLBumpShader

GLPhongShader

GLScene Objects

As described in ‘How does it work?’ chapter I refer to GLScene objects as objects not present in components tabs. GLScene objects are created, edited and deleted in GLScene editor. Objects are arranged into categories. Some objects don’t fall into any category and stand alone.

Let’s have a closer look at GLScene editor before we start describing objects one by one. GLScene editor is opened by double clicking on GLScene in object editor. A tree view contains all objects in your scene. At the beginning there will be only **Scene root** that contains **Cameras** and **Scene objects**. **Scene root** refers to *GLScene.Objects* which is the top element in scene hierarchy. **Cameras** have special position in the hierarchy. Right click on **cameras** then click on **Add camera** a new camera is created. Later the camera can be moved around the hierarchy tree and become a child of any other object. All other objects you create will be arranged under **scene objects**. To place an object in the scene right click **scene objects** and select the object you want. If you want the new object to become a child of already existing object right click on the parent object. A new object is always placed on the last position within the current hierarchy tree level.

The order of objects in the tree hierarchy is important. The topmost object gets rendered first. If it has any children they are rendered next, the one on the top first again. The rendering goes on branch by branch until it reaches the bottom of the tree. The rendering order is important for transparent objects. Transparent objects must always be rendered last otherwise you can get visual artifacts in the scene. Some objects though (like HUD objects or particles) take care about rendering order themselves and can be placed anywhere in the tree hierarchy. You can change object’s position in the tree by dragging it over to another object or by right clicking on it and selecting **Move object up** or **Move object down**.

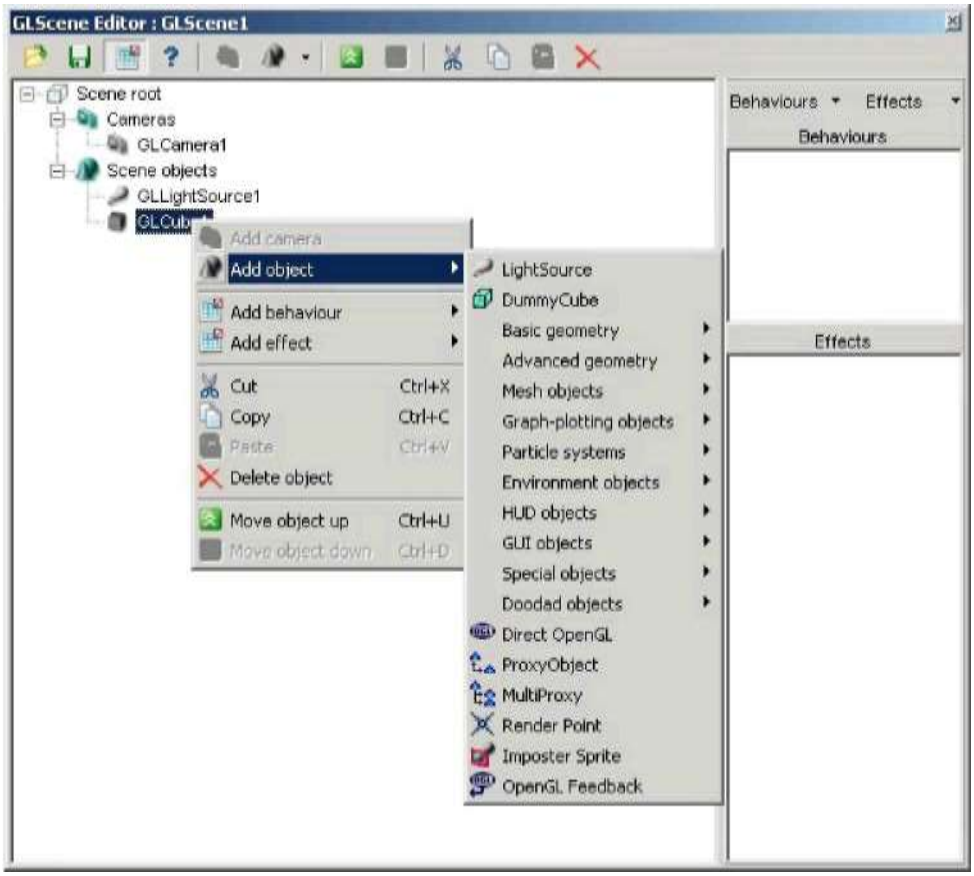
Common object properties

All objects have some properties that you will use frequently. Here is an itemized list of some properties common to all objects:

- ***Behaviors*** – If you add behavior to object, it will act according to that behavior. Usually they alter object's movement, sound or collision detections. You can add more than one behaviors. They work together with *GLScene Utils* components. List of behaviors:
  - Collision
  - Simple Inertia
  - Simple Acceleration
  - Sound Emitter
  - Movement controls
  - FPS Movement
  - DCE Static Collider
  - DCE Dynamic Collider
  - ODE Dynamic
  - ODE Static
  - ODE HeightField Collider

- ***Children*** – Not shown in object inspector. List of all children this object has. Children can be accessed by index starting with zero.
- ***Count*** – Not shown in object inspector. Number of children plus one. The array of children is zero based index.
- ***Direction*** – Vector pointing in 'front' of object. This vector is normalized. It means that it's length is 1. Default value is [0,0,1].
- ***Effects*** – Each object can emit particle effects. If you use one of PFX managers you can add effects here.
  List of available effects:
  - PFX Source
  - FireFX
  - ThorFX
  - ExplosionFX

- ***Objects Sorting –*** The order in which objects get rendered.
- ***Material*** – Described in *TGLMaterialLibrary* chapter.
- ***Parent*** – Not shown in object inspector. Object one level up in scene hierarchy tree.
- ***PitchAngle*** – Angle describing object's rotation around Y axis in degrees. Be aware that changing this value does not always produce the same effect. It is recommended to use *Direction* and *Up* vectors instead.
- ***Position*** – Vector describing position of object in 3D space. Default value is [0,0,0].
- ***RollAngle*** – Angle describing object's rotation around X axis in degrees. Be aware that changing this value does not always produce the same effect. It is recommended to use *Direction* and *Up* vectors instead.
- ***Scale*** – The size of object is set by this vector. You can scale the object non-uniformly, for example only along X axis. Children do not inherit scale value. Default value is [1,1,1].
- ***ShowAxes*** – Boolean value enables displaying colored lines aligned to object direction axes.
- ***TagFloat*** – Similar to well known *Tag* value only it stores single.
- ***TurnAng le*** – Angle describing object's rotation around Z axis in degrees. Be aware that changing this value does not always produce the same effect. It is recommended to use *Direction* and *Up* vectors instead.
- ***Up*** – This vector together with *Direction* is used to describe object's rotation in space. *Up* vector is always perpendicular to *Direction* vector and is also normalized. Default value is [0,1,0].
- ***Visibility culling*** – Visibility culling can be used to speed up rendering. Objects that are not in camera view are quickly discarded. But this works only in special cases. This visibility culling is only object based, not polygon based.
- ***Visible*** – You can hide or show object. Any code in object's progress event gets executed even if *Visible* is *false*.



TGLCamera

You can think of a camera as a point in space where the scene is viewed from. Camera has position, direction and up vectors same as other objects. You cam move and rotate camera around the scene. Simpler and more useful method of orienting camera is to make it look at another object. Assign the object to *Target* property. The camera will always point to that object wherever it moves.

FieldOfView is a value that changes camera lenses. Lower values make the angle overlooking the scene wider and higher values zoom to more focused area. Make sure you don't use any crazy values as the twisted perspective can be very disturbing for the user.

You should also know something about culling planes. Polygons that are too close or too far from the camera are not rendered at all. Planes that divide the scene in visible and invisible parts are called near and far culling planes. You can set the position of these planes with *NearFrustrumRange* and *FarFrustrumRange.* Normally only the far culling plane is changed. This creates one small problem. As the camera moves forward objects suddenly spring into the view as they get past the far culling plane. To reduce this unwanted effect fog is often used. You can enable fog in *TGLSceneViewer.Buffer* together with fog options. Fog has start and end range. Any polygons between the two borders will change transparency with full opacity at start range and full transparency at end range. Make the end range of the fog same as far culling range and color of the fog same as color of scene viewer background.

TGLLightSource

First we should explain what lights can do in GLScene. Without lights the scene would be dark. Lights illuminate the scene. We can have maximum number of eight lights. Every light except of parallel lights has a range limit how far it shines. Beyond that distance it has no effect. From the

position of the light to the maximum distance the light slowly diminishes. This is called light attenuation.

Lights do not cast shadows. If you have a sphere in front of the plane there will be no shadow on the plane, the light rays will seem to pass through the sphere without any interference. You have to use other techniques to obtain shadows. Lightmaps, Z-shadows or shadow volumes for instance.

There are three types of lights:

1. **Omni light** - this is a light located in one point radiating light rays equally to all directions. You can thing of omni light as of an electric bulb hanging on a cable freely in the air.
2. **Spot light** - spot light shines a beam or a cone of light in one direction. You can change the width or angle of the cone. A 360º cone would become an omni light. Example of spot light is a torch light.
3. **Parallel light** - uniform mass of parallel light rays with same direction shining from a plane in infinite distance form a parallel light. You can change position of parallel light but it has no effect whatsoever. Parallel light is usually used to simulate sunshine.

TGLDummyCube

GLScene also has helper objects. Helper object is an invisible object that is used for organizing other objects in groups, to show distances, directions and positions in space. You should learn how to use dummy cube as much as possible because it can simplify a lot of things. Dummy cube can be used to store objects as its children. If you want to for example move all of the objects just move the dummy cube. If you want to delete the objects just call *DeleteChildren* function of the dummy cube. Dummy cube can simplify rotations. You can create joints with dummy cubes where each dummy cube will have restricted rotation only in one axis.

You can make dummy cube visible run time with *VisibleAtRuntime* property. Dummy cube is represented by cube outlined with dotted lines.

Basic geometry

Basic geometry objects are simple objects with hard coded geometry. The geometry can be described with mathematical formulas. You can change individual parameters of the object but the basic shape will always stay the same.

TGLSprite

Sprite is a 2D image mapped on a plane that always faces the camera. Unlike HUD sprite *TGLSprite* is placed in 3D world and can be moved closer or further from the camera. If the sprite gets too close to the camera it can create unwanted visual errors. A transparent texture is usually used with sprites. The image can turn around its center by changing *Rotation* property.

TGLPoints

You can create array of points or just one point with this object. Points are dots in space. They have *Size* parameter which determines how many pixels in diameter the point will have on screen. If the point has size 3 it will be represented by 3 pixels on screen whether the point may be close to the camera or far away from the camera. Perspective does not apply here. Points can be rounded or squared and antialiased.

TGLLines

Lines are basically points connected by lines. Points are called nodes here. Lines have *Width* with same perspective restrictions like points. Nodes can be visible or not and can be represented by cubes, stars or spheres with changeable size.

* **TGLPlane**

Plane is a single quad. Quad is a polygon composed of four vertices and two triangles. Plane has *Width* and *Height* values. *Direction* of the plane is important. It points in the direction of normal of the plane. Normal is a vector that tells us which side if front and which side is back. If we look at the plane from the front side we can see it, if we look from the back side we can't see it. You will be able to see the plane only if it's direction will point towards the camera.

Vertex shading can sometimes create problems with large planes. If edges of the plane are stretched out too far from the reach of lights the plane will become dark as a whole.

TGLPolygon

TGLCube

Cube is composed of six planes, each plane can be disabled. Cube has *CubeWidth*, *CubeHeight* and *CubeDepth* properties.

TGLFrustrum

Frustrum is a twisted cube with variable size of top and bottom sides.

TGLSphere

Radius is the size of the cube, *Segments* specifies how many polygons will be used to draw the cube and how smooth the curvature will be.

TGLDisk

TGLCone

TGLCylinder

TGLDodecahedron

TGLIcosahedron

Advanced geometry

More complicated than basic geometry but the description is the same.

TGLAnimatedSprite

Animated sprite is based on sprite only it displays animated sequence. The animated sequence is stored in single bitmap. The bitmap is divided in rows and columns. The grid contains individual frames for the animation. This technique is great for smoke or explosion effects but you have to make the bitmap yourself.

TGLArrowLine

Arrow line is a combination of cylinder and cone. You can change the width and height of both. It points in *Direction* vector.

TGLAnnulus

A hollow tube.

TGLExtrusionSolid

TGLMultiPolygon

TGLPipe

TGLRevolutionSolid

TGLTorus

Mesh objects

Mesh is a pool of polygons defined with vertices, texture coordinates and connectivity information. Group of mesh objects in common has same feature: it's geometry has to be loaded from a file. Such file is usually created by 3D modeling programs like 3DS MAX, Maya or Milkshape. Supported formats are 3ds, obj.

TGLActor

This is an object you will use if you want geometry that will change shape in time. Living creatures are good example. Actor has same base as *TGLFreeForm* but it is optimized for moving. Animations have to be loaded to *Animations.* Actor needs *AnimationControler* component. You can use Quake md2 format. This format contains set of meshes in key frame positions. Subsequent frames are then interpolated between the key positions. This format is fast and suitable for low poly models. Another format is Halflife smd. It uses skeletal model animation. Actor has skeletal structure with vertices attached to bones. The animation is described by recorded angle values between individual bones in time. You can combine different animations at one time. Following animations can have interpolated transitions.

TGLFreeForm

This object is used very often. It is capable of loading various mesh formats. To be able to use some format you have to add appropriate loader unit. For example if you want to use 3ds format you have to add GLFile3DS unit to uses clause. Call *LoadFromFile* function to load the geometry. Texture coordinates are usually included in the file. Some formats support using of multiple textures for one object. You just have to set the textures to the *Mesh* list. To successfully load the texture the geometry has to be loaded before texture.

Example

Q:How to copy one or more GLFreeForm Object from one GLScene to another at runtime ? And What is the best method to copy a GLFreeForm in the same GLScene at runtime?
A:Hi you can try like this code

```
var
  NewCube : TGLCube;
  //NewSphere : TGLSphere;
  // NewFreeForm : TGLFreeForm;
  SelectedObj, TargetObj : TGLBaseSceneObject;
begin
 SelectedObj := GLCube1; // Object to copy in GLScene 1
 TargetObj := GLScene2.Objects; // GLDummyCube2 // Parent Target object in
GLScene 2 or 1

 if SelectedObj is TGLCube then
 begin
   NewCube := TGLCube.CreateAsChild(TargetObj);
   NewCube.Assign(TGLCube(SelectedObj));
 end;
 //else
 //if SelectedObj is TGLSphere then
 //begin
 //  NewSphere := TGLSphere.CreateAsChild(TargetObj);
 //  NewSphere.Assign(TGLSphere(SelectObj));
 //end
 //if SelectedObj is TGLFreeForm then
 //begin
 //  NewFreeForm := TGLFreeForm.CreateAsChild(TargetObj);
 //  NewFreeForm.Assign(TGLFreeForm(SelectObj));
 //end
 // etc....
```

TGLMesh

Settings

Sometime objects are closed to each others so when viewing form far rendering they can be overlapping with scratches. To exclude it use the properties

The right way is to work with next properties for Viewer/Camera:
DepthOfView := 1000 - 100000;
NearPlaneBias:= 1; (* decreasing NearPlanesBias to 0.001 the issue defects increases. Using NearPlanesBias to 10 the quality improves a lot *)
Buffer.DepthPrecision = dpDefaults (* *dp32bits* if not a lot of issues with OpenGL is running in Intel embedded GPUs *)
FreeForm or Parent object.ObjectSorting --> usually setting to osRenderFarthestFirst
(* FreeForm or Parent object..VisibilityCulling --> try set to vcObjectBased or vcHierarchical and you can try to set in Viewer.Buffer.ContextOptions this param : * roTwoSideLighting *)
Buffer.DelpthTest := True;

```
Buffer.FaceCulling := True;
```

TGLTilePlane

TGLPortal

TGLTerrainRenderer

To render outdoors landscape use terrain renderer. The advantage of terrain renderer is LOD optimization. LOD takes care of **L**evel **O**f **D**etail in the scene. Terrain is composed of quads. Quads closer to the camera are denser with high detail. Quads further from camera stretch across larger areas loosing the detail. This results in less polygons to be rendered without loosing visual quality. Data for terrain renderer is provided by HDS components described in 'components' section. Default *Up* vector of terrain renderer is Z axis.

Graph-plotting objects

Object in this group are often used for scientific visualizations. Their shape is described with mathematical equations.

TGLFlatText

TGLHeightField

TGLXYZGrid

Displays space grid along X,Y and Z axis. Individual axis can be turned on and off. Size of grid tiles can be changed.

Particle systems

Particles are small sprites in large numbers generated according to same rules. Particles create special effects like fire, smoke, explosions, rain or snow.

TGLParticles

TGLPFXRenderer

PFX renderer is a helper object which has to present in the scene so any object can have *Effects* properly functioning. The system works like this: PFX manager describes settings for the effect, scene object has the effect in *Effects* list and PFX renderer takes care about rendering particles. This seems to be too complicated but it is very flexible.

Environment objects

By default background color of the scene is light gray. The color is defined by *SceneViewer.Buffer.BackGroundColor* property. If you want other background than uniform color use environment object.

TGLEarthSkyDome

TGLSkyDome

Sky dome creates gradients of color stripes along horizontal plane. You can make as many stripes as you want. You can add small dots in *Stars* list. Stars can even glitter.

TGLSkyBox

Sky box is a large cube with inverted normals and six textures covering whole skyline. The textures must align each to another and create panoramic illusion.

HUD objects

HUD is **H**eads **U**p **D**isplay. These objects are rendered as bitmaps on screen. Their position is defined only by X and Y values. X is horizontal and Y is vertical position of HUD object. The origin [0,0] point is upper left corner. You do not have to care about rendering order of HUD objects (only among each other), they always get rendered last.

TGLHUDSprite

Universal HUD object. Displays bitmap on the screen. Use material library to store the texture. Transparent textures are often used. Texture should have power of two dimensions. You can rescale the size of the object with its *Width* and *Height* values. HUD sprite can turn around center with *Rotation* property.

TGLHUDText

This HUD object displays characters in its *Text* property. How is the text displayed is determined by GLScene font component.

GUI objects

GUI objects are described in *TGLGUILayout* section.

GUI objects

TGLRootControl

TGLPopupMenu

TGLForm

TGLPanel

TGLButton

TGLCheckBox

TGLEdit

TGLLabel

TGLAdvancedLabel

TGLScrollBar

TGLStringGrid

TGLBitmapControl

Special objects

These objects don’t fall in any other category.

TGLLensFlare

Lens flare is artifact that is caused by camera lenses looking in strong light in real world. This adds more realism in your scene. Lens flare is normally positioned in same location as light source. Rings and streaks are created when looking straight at lens flare. Both can be changed in number, size and quality. Lens flare naturally fades away when hidden behind other object or out of view.

TGLTextureLensFlare

TGLMirror

TGLShadowPlane

Shadow plane is a fast technique how do render dynamic shadows. Shadows are projected on a flat plane only. You have to enable stencil buffer of scene viewer. Assign light source that will be used to generate shadow and shadowing object. The shadowing object and all of its children will

cast shadows on Z shadows plane.

TGLShadowVolume

TGLZShadows

TGLTextureEmitter

TGLProjectedTextures

TGLBlur

TGLTrail

Doodad objects

Special category, kind of more advanced geometry.

TGLSpaceText

This object displays text as three dimensional letters. Regular Widows fonts are used. Text can have variable depth or *Extrusion.*

TGLTeapot

Teapot is an object well known from 3DS MAX studio.

TGLTree

Realistically looking trees can be created by this object. There is a lot of settings which can make the tree look like any type of tree. Tree is composed of one main trunk with diverging branches and leaves. Number of branches (and polygons) is set by *Depth* property. Leaves are represented by planes covered with front and back textures.

TGLWaterPlane

This object creates good looking realistic water.

Other objects

TGLDirectOpenGL

Execute your own OpenGL commands. You have to know OpenGL syntax to use this.

TGLProxyObject

In cases when you have a lot of same objects, especially *TGLFreeForms,* it is good practice to use proxy objects. You will gain a lot in performance. Proxy object refers to *Master* object. Geometry of master object is rendered in place of the proxy object. Proxy object has its own position, direction and scale. The geometry is altered accordingly.

TGLMultiProxy

Multi proxy object is an advanced version of proxy object. It has more masters that are chosen according to distance from camera. This technique is used with set of objects with different number of polygons but resembling each other. Low detail objects are displayed in distance and high detail objects are displayed close to the camera.

TGLRenderPoint

TGLImposterSprite

Imposters are described in *TGLStaticImposterBuilder* section. Imposter sprite is the object that is rendered in the scene.

TGLOpenGLFeedback

Runtime object creation

Until now we spoke only about design time functions and features of GLScene. As you will learn more about GLScene you will find out that creating objects runtime is inevitable. Components are created the usual Delphi way with *Create* directive. But scene objects are created differently. Those objects have to be registered in the scene clearly saying where the objects position in the scene hierarchy is.

If we want for example create a sphere named *MyGLSphere* as a child of *GLCube1* the correct code is like this:

```
var
MyGLSphere: TGLSphere

begin
     MyGLSphere := TGLSphere(GLCube1.AddNewChild(TGLSphere));
end;
```

This way the new sphere will be placed correctly in the scene as a last child of *GLCube1*. If you want to make it a first child of *GLCube1* call *AddNewChildFirst* instead. Any object can be moved in the scene hierarchy with *MoveUp* and *MoveDown* functions or by changing *Parent* property. To destroy the object call *MyGLSphere.Free* function.

Creating new class

You will also find very useful to inherit your own classes from existing ones. This process has same rules like any other Delphi object. Let's make a new class for our spinning cube.

```
type
TGLSpinningCube = class(TGLCube)
private
     FSpinSpeed: single;
published
     property SpinSpeed read FSpinSpeed write FSpinSpeed;
public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
    procedure DoProgress(const progressTime : TProgressTimes); override;
end;

constructor TGLSpinningCube.Create(AOwner : TComponent);
begin
     inherited;
     FSpinSpeed := 1000;
end;

destructor TGLSpinningCube.Destroy;
begin
     inherited;
end;

procedure TGLSpinningCube.DoProgress(const progressTime : TProgressTimes);
begin
```

```
     inherited;
     TurnAngle := TurnAngle + progressTime.DeltaTime * FSpinSpeed;
end;
```

The on progress event is different at this level. *ProgressTime* is used to access *DeltaTime*. You also have to call *inherited* directive otherwise the object would not progress at all. To create object defined by this class you would use technique described earlier in this chapter or register the class in GLScene and add it with help of GLScene editor.

Case study: 3D Tetris

In this chapter I am going to give you my ideas about making a simple Tetris game with GLScene. I am not going to write only some code, I will mainly focus on structure and design of the game. I am going to keep it as simple as possible. After all this is beginner's guide. I am sure many people would do much better 3D Tetris so don't take my word that this is the best way how to make a 3D Tetris game.

I don't have working code for the game. This is a kind of mini game design document. I would be interested if anyone makes the game real. Please send it over to me and I will publish it with this book.

Design

To make it ultra-simple we are going to create a clone of old Tetris. The third dimension here is going to add only a little bit of eye candy. It is actually going to be '2.5D Tetris'. Camera will look at the scene from an interesting angle and perspective. That's all. The game will take place in two dimensional grid just like old Tetris. Only the blocks will be made up of cubes instead of squares.

There will be no menus. The game will start right after the program is launched or it will wait for the user to press a key. During game a label will show number of achieved points. After the game is finished a message window will show information that the game is over and player's score. The program will then terminate.

In classical Tetris the number of types and shape of blocks is defined. We are going to randomly generate shape of the block on its creation. The block will have constant descending speed. The player can move with them left and right and rotate them clockwise and counterclockwise. Input will be handled by keyboard.

Structure

We are going to need *TGLScene*, *TGLSceneViewer*, *TGLCadencer* and *TGLWindowsBitmapFont* components. The playing field will be 11 units wide on X axis and 21 units high on Z axis. We will place the camera on negative Z axis overlooking the whole scene from top right side. One *TGLPlane* will form the ground at position [0,0,-0.5] and two tall *TGLCube*s will be standing on sides to set borders. At positions [-6,0,10] and [6,0,10], 10 units high. All of these objects together with nice camera position can be made design time. Don't forget to add light to the scene. You can place it wherever you like.

We are going to use our game logic for detecting collisions of the falling cubes. The playing field will be represented by array of singles:

```
var
FloodLevels: array[-5..5] of single;
```

Each member of this field represents how high the stack of boxes reaches during the play. Every time the block will be securely placed in final position corresponding field will be set to new height level. This will happen only **after** the block will rest at the bottom, not during his fall. There are no cubes at the beginning of the game so initial values are zero. Values stored in *FloodLevels*

describe centers of cubes. Actual edges of cubes will be 0.5 units higher but it will be considered in collision detection.

To detect user input we will use *IsKeyDown* function contained in *Keyboard* unit. This is simple method how to capture keyboard input. We will define five possible user actions:

```
T3DTInput = (inpNone,inpMoveL,inpMoveR,inpRotateL,inpRotateR);

var
UserInput: T3DTInput;
```

Keyboard status will be captured in cadencer progress event and stored in global variable *UserInput*.

```
UserInput:=inpNone;
if IsKeyDown(VK LEFT) then UserInput:=inpMoveL;
...
```

It needs to say a few words about capturing keyboard and mouse input. Always use necessary minimum of code to capture the controller input in global variable. Especially *OnMouseMove* event can cause big slowdown if you write too much code in it. Windows consider mouse movements first priority and try to execute its code before other procedures. If you write complicated code in this event the system becomes flooded with this code trying to execute it with the smallest mouse movements. Not enough computing power remains for cadencer progress. All you should write in *OnMouseMove* is *MouseX := X; MouseY := Y;*

Information about current score is updated every cadencer step with *TGLHUDText* object placed in upper left corner.

T3DTBlock class

The core of the program is going to be our new class derived from TGLDummyCube. This class will represent individual blocks. The dummy cube will take care of movements and rotations of the whole block. It will have *TGLCube* children as boxes in the block. Each cube will compare its Z position with *FloodLevels* to detect collisions. We are going to use *Round* function to snap the cubes position on X grid.

The block will be initially created at position [0,0,0]. One central TGLCube will be added to every block again at position [0,0,0]. Random number of cubes will be added in four directions. The block will form a cross, L-shape or line. Five cubes in a row maximum. Cubes should be 0.95 units in size and positioned at [1,0,0], [2,0,0], [0,0,1], [0,0,2], [-1,0,0] and so on. All cubes will have Y position 0 at all times. After all cubes are created the block is repositioned to starting position well above playing field with random X position. Boolean value called *Active* will represent status of the block. At beginning block will be active (descending), once it will land on ground or other blocks we will set *Active* to false.

If the cube is active it is moved tiny bit down the Z axis during block's progress event: *Position.Z := Position.Z – deltaTime * speed*. Then we check for collisions. We have to process all children and compare their absolute positions with both sides (X<-5 and X>5) and bottom (Z<0) of the playing field. We also have to check against other blocks already in the playing field:

```
var
absPos: TVector;
```

```
setVector(absPos, Children[i].AbsolutePosition);
if (absPos[2] - 1) < FloodLevels[Round(absPos[0])] then …
```

This code fragment detects if the cube iterated by *Children[i]* is lower then value stored in *FloodLevels* field. Absolute position of the cube is stored in *absPos* variable. This is important because absolute position is real position in world coordinate system while *Position* is position relative to the parent. Which member of *FloodLevels* field is to be compared against is determined by rounding absolute X position of the cube. If the cube rests on ground or other cube, *Active* property is set to false, we add points to score and program is informed that another cube can be created.

To respond to user actions we check which *T3DTInput* status is active and do the task:

```
case Form1.UserInput of
     inpMoveL: Slide(-1);
     inpMoveR: Slide(1);
     inpRotateL: Pitch(-90);
     inpRotateR: Pitch(90);
     end;
```

Slide and *Pitch* functions are GLScene functions to move and rotate objects. Now we should check for collisions again and undo the movement or rotation if the new position of block is not valid.

The last thing is to detect if the game is over. This can be done in cadencer progress event. We can simply check *FloodLevels* if any value is over 20. If it is so disable cadencer and display massage with number of points achieved and game over information.

Homework

This is it. With some tweaking that is all you need to make Tetris game. I chose this example to show you how powerful GLScene is. Not only that you can set up all game environment design time. You take advantage of functions that move and rotate objects together with their children. The code to copy this behaviour would be quite complex.

In the last paragraph I will give you more ideas to think about to make the game better.

- Make some interface. The program needs menus desperately. Use standard Delphi or GUI components.
- Let the game be restartable with levels and increasing difficulty – game speed.
- Make real collisions with collision manager and let the blocks rotate and slide smoothly. That would require more advanced solutions: can the block collide while rotating?
- Add sound and music. Player should hear sound feedback for his actions.
- Make it customizable – graphics, sound, keyboard options, size of playing field and blocks.
- Make score table with best results together with player's name that is saved.
- Make it look prettier. Put different textures on blocks. Add particle effects, sky box, lens flare. The game can take place in a canyon with bridge above. Passing cars or trains would drop blocks. Or King-Kong on top of two skyscrapers can throw the blocks down. The limit is only your imagination.
- Add the third dimension to the game play. But be careful! Many 3D remakes of old 2D games were not successful because they became too complicated and difficult to play. You should thing carefully about camera controls. The game can pause during camera

repositioning. Make some blocks transparent. Add *TGLXYZGrid* for easier orientation.

- Write narrative story for Tetris – **now that is a challenge!**

Let me give you one last advice. There is more to programming than just clean code. Don't look only for technical documentation like this one to make your programs better. There is a lot of articles about programming theory other than technical on Internet. Users now are choosier. Your application has to be more user friendly than other applications of same kind. Even if you are making freeware you still want users to be happy about it. You should give same time to designing and debugging as to programming itself. It pays of!

Links

GLScene home page www.glscene.org

If you have any reasonable question, go ahead and ask at GLScene forum. If the question is not stupid you will get qualified answer in short time.

GLScene forum https://sourceforge.net/p/glscene/discussion/

Conclusion

If you've read this far you should have basic understanding how is GLScene organized, what it can be used for, what are its parts and some programming techniques specific for GLScene. Now I suggest you go back to GLScene demos. If you failed to understand what is going on before, you should have the idea now. It is sometimes difficult to find the function or procedure that you are looking for among the myriad of GLScene features. But all functions have reasonable names and soon you will find those you need for your purpose. After you are sure you can follow the demos you can start to create your own applications.

Index

References

1. OpenGL Programming Guide. Eighth Edition. The Official Guide to Learning OpenGL version 4.3.

2. Randi J. Rost (2004). OpenGL Shading Language. ISBN 0-321-19789-5.