# *Geometry Primitives*

Sandro Spina
Computer Graphics and Simulation Group

Computer Science Department
University of Malta

# *The Building Blocks of Geometry*

- The objects in our virtual worlds are composed of a number of basic primitives.
- With only these few building blocks, we can create highly complex realistic structures.
- During this module you will learn most of the things you need to know in order to draw objects in three dimensions from these building blocks.
- In this module we shall be using OpenGL to describe and carry out some implementations using these primitives.

# Points ... (not Pixels)

- We have already seen that our image will finally be rendered as a number of pixels.
- A pixel is the smallest element in your computer screen ... And can be one of many different colours.
- ... So if you want to draw a line, just select two pixels and fill in all the pixels falling on the line between them.

- In Computer Graphics however, our points are fundamentally different.
- We don't care (as such) about screen coordinates but rather positional coordinates in some volume.

# A Drawing Canvas (in 3D)

- We need a 3D coordinate system to define out 3D canvas.
- On this 3D canvas we'll draw our graphics primitives.
- We can visualise a 3D canvas as a volume with a Cartesian coordinate space enclosed.
- We define boundaries on this space ... Let`s say -100 to +100 along the three axes x,y and z.
- This volume will determine the clipping volume.

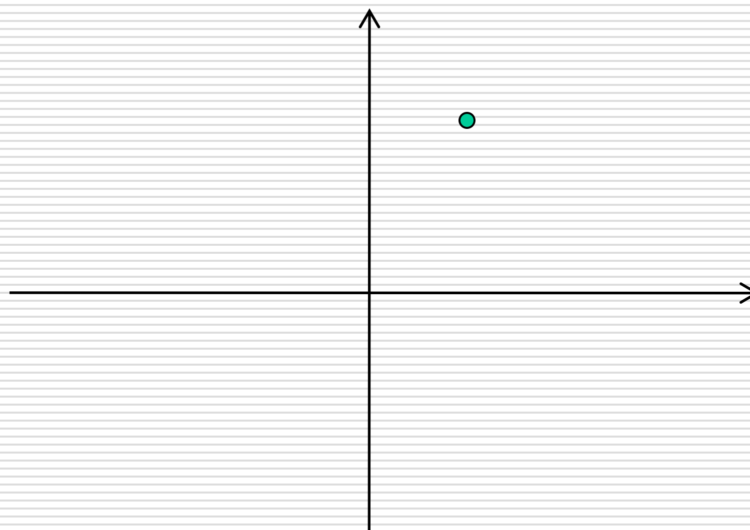
# The Vertex (3D Point)

- The most basic of all primitives is the point (or vertex)
- Remember the game (on newspapers mainly) connect the dots ...
- The image starts to make sense only when you connect the dots (points)
- A set of points define the image .... But not just that
- How they are connected plays a fundamental role

- In order to draw a point (using OpenGL here but this is valid with DirectX as well using different syntax) we use the glVertex3f function.

# *The OpenGL Vertex (Example)*

- glVertex3f(20.0f, 40.0f, 0.0f);
- Note that we use floating point numbers to represent coordinates.



# The Primitives ….

- Important: A primitive is the interpretation of a set of vertices into some shape.
- A vertex on it's own can be a primitive but we need to specify that the point is actually a primitive.
- However a vertex can be part of some other primitive ...
  - The endpoint of a line
  - The corner of a quad
  - The corner of a triangle
- In OpenGL there are 10 primitives – as we'll see most of the times however we'll be using the triangle primitive.

# *Drawing Primitives ....*

- We need to indicate to our graphics hardware that we want to start drawing a primitive ...
- In OpenGL we use the function glBegin() to tell OpenGL to begin interpreting a list of vertices as a particular primitive.
- Once your program submits all the vertices a call to glEnd() will advice OpenGL that you're done ...
- If you've done everything right OpenGL will then be able to display that primitive on screen.

# *Drawing Points ….*

- As we said earlier a Point on it's own is also a primitive if we want to.
- The following code is telling OpenGL that we want to draw point primitives …

```
glBegin(GL_POINTS);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(20.0f, -20.0f, 0.0f);
    glVertex3f(-20.0f, 20.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, -20.0f);
glEnd();
```

# Check First Example (Points)

- I am using code::blocks to carry out my implementations of OpenGL.
- I recommend you download it. It's fantastic.
- The example we'll be checking out is available on the OpenGL Super Bible 4th Edition book.
- There are a number of functions which we have not covered ... Don't worry too much about them for the time being. Focus on the drawing of the point primitives.
- Switch to Code::Blocks example points spiral

# Properties of Points

- Points are draw by default as squares ...
- We can specify the size of the points using the function glPointSize() ... By defauly value is 1.
- Note that the point size cannot be changed between the glBegin() and glEnd() calls ....
- We need to set this before.

# The Line Primitive

- The next logical steps is to draw a line primitive.

- As opposed to a GL_POINTS primitive which simple draws a point/vertex specified, GL_LINES takes two vertices and draws a line between them.

- 
```
glBegin(GL_LINES);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(20.0f, 20.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(-20.0f, -20.0f, 0.0f);
glEnd()
```

# Check 2nd Example (Lines)

- In this second example we'll be drawing lines from the origin to different parts of the circumference of a circle.

- Again cos(angle) gives us the x-coordinate
- And sin(angle) gives us the y-coordinate

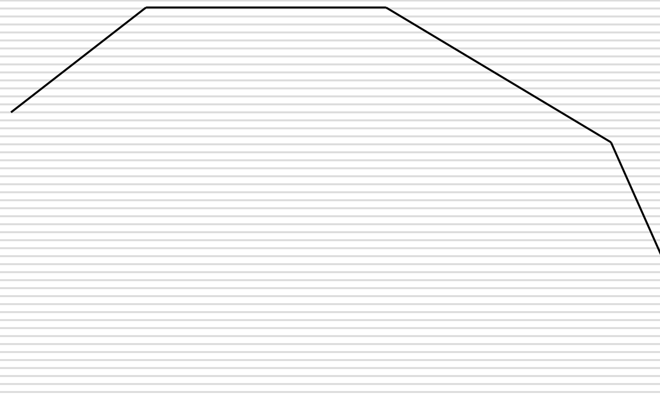
- We keep the z-coordinate to 0 ... But we can easily extend this example to draw a sphere.

# Line Strip & Line Loop Primitives

- Two different OpenGL primitives ...
- In GL_LINE_STRIP a line is drawn from one vertex to the next.
- Similar to GL_LINE but with reuse of vertices as opposed to GL_LINE.
- The GL_LINE_STRIP can be composed of as many vertices as required.
- GL_LINE_LOOP behaves exactly like GL_LINE_STRIP but draws an additional line between the first and last vertex specified. Perfect to draw polygons.

# Line Strip & Line Loop Primitives

- We can use GL_LINE_STRIP to approximate a curve.



- Clearly the more vertices specified (the more close to one another) the better the approximation is.
- Revisit example one but now using GL_LINE_STRIP instead of GL_POINTS

# Line Stippling .. (not a primitive)

- Stippling is a property of the line ....
- Essentially makes the line appear either dotted or dashed.
- glEnable(GL_LINE_STIPPLE) enables this property whereas glDisable(GL_LINE_STIPPLE) disables it. In OpenGL properties are set and reset in this way.
- A call to glLineStipple(Glint factor, Glushort pattern) establishes the pattern used to draw the lines.
- The pattern parameter is a 16-bit value, where each bit represents a section of the line (on or off). Factor parameter is used as a multiplier of the pattern.

# The Triangle Primitive (i)

- Without any doubt the most important primitive in CG

- So far with the primitives we've seen we can draw everything especially if you consider that ultimately a triangle is simply 3 lines where the third vertex is connected back to the first.

- The problem with lines is that you are not creating surfaces ... Hence we apply colour. We can of course apply colour to the line itself but not the surface between the lines.

- With lines we can create **wireframe** cube and not a **solid** cube.

- To draw solid objects we need polygons, and triangles are the most basic of all polygons.

# The Triangle Primitive (ii)

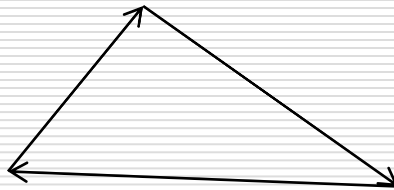
- To draw the simplest of polygons we make use of GL_TRIANGLES

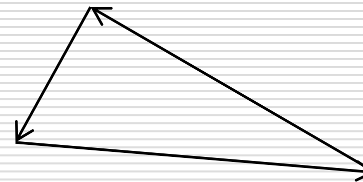
- ```
  glBegin(GL_TRIANGLES)
      glVertex2f(0.0f, 0.0f);
      glVertex2f(20.0f, 20.0f);
      glVertex2f(10.0f, 10.0f);
      glVertex2f(-20.0f, 0.0f);
      glVertex2f(0.0f, 20.f);
      glVertex2f(-10.0f, 10.0f);
  glEnd();
  ```
  ```

# *Winding (i)*

- You might have noticed that there are two ways by which I can specify the same triangle.

- Either V0, V1, V2 or V2, V1, V0





- This very important property of polygons is referred to as **winding** ... and essentially determines which direction the 'top' surface of the polygon is facing. Very important for back-face culling.

# *Winding (ii)*

- The three vertices of the triangle make up its face which can be either front face or back face.
- In OpenGL, by default, polygons with counter-clockwise winding are considered to be front facing.
- In DirectX, by default, polygons with clockwise winding are considered to be front facing.
- Why is this important to know?
  - Front and back might have different characteristics
  - Back-Face Culling
- VERY IMPORTANT : consistency throughout

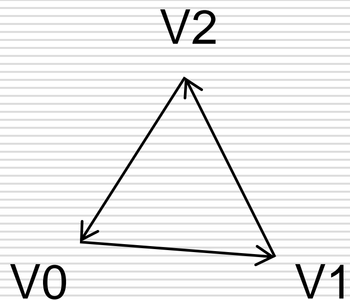
# Triangle Strips Primitive

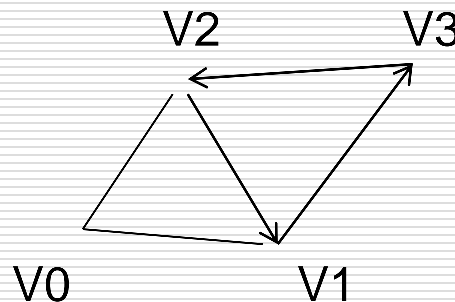
- To represent different shapes (models) we need to draw many triangles connected to each other.
- For example we can draw a square by connecting two triangles one to the other. We can then create a cube from 2*6faces = 12 triangles.
- Instead of specifying the vertices of the triangles for each triangle (in a similar fashion to LINE_STRIP) we can specify a sequence of vertices forming the triangles.
- GL_TRIANGLE_STRIP primitive.

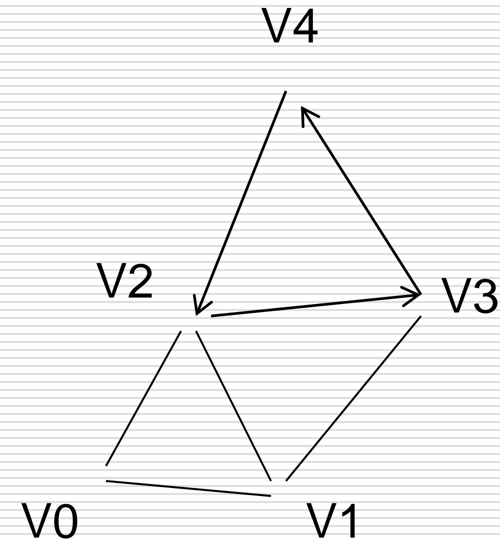
# An example …

- Let’s say we specify **five vertices** which will give us **four triangles**. I.e. One point per triangle after the first three (very efficient!!)

- V0, V1, V2, V3, V4



# Triangle Fans

- We can use GL_TRIANGLE_FAN in order to represent a polygon made up of a number of connected triangles all sharing a central point ...
- Assume we have V0, V1, V2, V3, V4 ....
- V0 is the central vertex ... Then we have triangles
  - V0, V1, V2
  - V0, V2, V3
  - V0, V3, V4
- Triangle fans are particularly useful to create cones .... Obviously the higher the number of vertices the smoother the circular base of the cone.

# An Example .... Using Triangles

- We’ll now be checking some OpenGL code which draws a cone ... (code available on the OpenGL SuperBible Book)
- We’ll be setting colours for the triangles faces
- Some of the functions we’ll see in the example we’ll be looking at in a few weeks time.
- Depth Buffers and Back-face Culling
- Switch to code::blocks to check the code.

# Quad Primitives ...

- To move from triangles to quads we simply need to add a further vertex ....
- GL_QUADS draws a four-sided polygon which does not necessarily have to be a square
- Quads are used to define flat surfaces like triangles.
- However it is important (very) that these vertices lie on the same plane. This was not important for triangles because any three points define one plane. With quads it's different since we use four points.
- Finally quads have clockwise winding !!!

# Quad Strips

- Behave in a similar way to lines and triangle strips ...
- 6 vertices define two quadrilaterals...
- 7 vertices define ? Nothing .... Because for every new quad you need to define to further vertices.
- Therefore 8 vertices = 3 quads.
- Note that winding remains clockwise for GL_QUAD_STRIP

# Finally ... GL_POLYGON

- The most general primitive ... very rarely used (if not never) is the GL_POLYGON
- You can use it to draw a polygon with any number of sides.
- Actually it is quite redundant ...
- All vertices of the polygon must lie on the same plane.
- To make sure this is the case you can always use GL_TRIANGLE_FAN ... In this case only the triangles will need o be in the same plane and that's guaranteed !!

# Convex Polygons ....

- For rendering performance reasons it is important that polygons are convex.
- With triangles this is not an issue (well unless you align the three points)
- In general a polygon is convex if it does not have any indentation ... In other words if we had to draw a line over the polygon this line would not enter and leave the polygon more than once.
- If this line exists then the polygon is concave.

# Subdivision .... And edges

- We can try to transform a concave polygon into a convex one by adding some edges within the polygon. This can be done through a technique called subdivision of surfaces.
- Consider a polygon with the shape of a star. This is concave.
- However if this polygon is split up into further polygons it will become a list of convex polygons ...
- Actually it would be great if we could come up with a set of triangles to describe the polygon.

# Surfaces ....

- We have so far the building blocks of geometries ....
- Effectively what we'll be doing is building surfaces using these building blocks ...
- Quads and Triangles already do this ...
- We shall now be looking at a number of geometric shapes which are usually built into 3D APIs as quadratic functions.
- These functions allow us to draw spheres, cylinders, cones and discs.

# Quadratric Shapes / Surfaces…

- We shall again be using opengl (using C) to illustrate the examples however equivalents are available in DirectX.
- Quadratic shapes are rendered using the quads or triangle primitive.
- One can create many different objects simply by connecting together spheres, cylinders (or cones) and discs.
- Consider drawing an arrow with one cylinder, one cone, one disc.

# OpenGL Code Specifics – Creating a Quadric

- GLUT Library

```
GLUquadricObj *pObj;

pObj = gluNewQuadric();

//set parameters ...

gluDeleteQuadric(pObj) // free memory
```

# Quadric Shapes … Draw Styles

- GLUT allows us to modify the way these quads are rendered through the function :
- gluQuadricDrawStyle(GLUquadric *obj, Glenum drawStyle);

- We are offered four possible ways to render the quadrics:
  - GLU_FILL
    - Solid objects
  - GLU_LINE
    - Wireframe objects
  - GLU_POINT
    - Point set
  - GLU_SILHOUETTE
    - Adjoining edges are not drawn.

# Quadric Shapes … Properties

- Normals
  - Using gluQuadricNormals(GLUQuadricObj *obj, Glenum normals)
  - 2nd parameter specifies where to compute the normals. GLU_NONE, GLU_FLAT, GLU_SMOOTH.
  - Using gluQuadricOrientation() we can also specify the orientation of the normals.
- Textures
  - Using gluQuadricTexture(GLUQuadricObj *obj, Glenum textureCoords) we can decide whether texture coordinates are generated on the quadric.
  - GL_TRUE or GL_FALSE

# Sphere ....

- The function call

  gluSphere(GLUQuadricObj *obj, Gldouble radius, Glint slices, Glint stacks)

- Spheres can be drawn as rings of triangles or quads.
- You can think of slices and stacks as the lines of longitude (vertical) and latitude (horizontal) respectively.
- Clearly the more slices and stacks the smoother the sphere will be ... Obviously we'll be defining more points thus making the sphere more accurate.

# Cylinder ….

- The function call

  gluCylinder(GLUQuadricObj *obj, Gldouble baseRadius, Gldouble topRadius, Gldouble height, Glint slices, Glint stacks)

- Again cylinders can be drawn as rings of triangles or quads. 1 quad = 2 triangles.

- You can think of slices and stacks as the lines of longitude (vertical) and latitude (horizontal) respectively.

- A the topRadius of 0 transfors the cylinder into a cone.

# Disk ….

- The function call

  gluCylinder(GLUQuadricObj *obj, Gldouble baseRadius, Gldouble topRadius, Gldouble height, Glint slices, Glint stacks)

- Again cylinders can be drawn as rings of triangles or quads. 1 quad = 2 triangles.

- You can think of slices and stacks as the lines of longitude (vertical) and latitude (horizontal) respectively.

- A the topRadius of 0 transfors the cylinder into a cone.

# Putting it all together ...

- Let’s have a look at a program which simply creates and puts together a number of these shapes in order to build a more complex shape.
- Switch to code::blocks
- It is very important that you realise that I’m using C and OpenGL just to make sure you understand that the language we use is irrelevant ... Many (if not all) computer graphics basic concepts are true irrespective of the language we use.

# Vertex and Index Buffers

- So far we have not been very efficient with our declaration of vertices ...
- Using a vertex buffer we can use memory on the GPU to store vertex information ...
- Then using an index buffer we can describe the topology of the geometry using the vertices (unique) which are stored in the VBO.
- This is very similar to what we've been doing ... But instead to sending primitive information to the GPU with each draw we store that information directly on the GPU.

# A note on terrain generation

- A flat terrain can easily be represented as 2 triangles ... Everyone would agree.
- But ... How do I create a terrain which is rough?
- Instead of representing my terrain with just 2 triangles I can represent it (still flat) as 10, 100, 1000, etc triangles.
- Then I can alter the y-coordinate value for each vertex in order to create a bumpy surface geometry.
- Clearly the higher the number of vertices the higher the granularity of my terrain.

# *Switch to DirectX using XNA*

- Let us now take a look at a program (available through the XNA creators site) which displays primitives.
- So far we have defined the vertices of the torus ... But that’s only good if we want to just draw the point set forming the shape.
- We also need to specify how the points are connected (triangle set) in order to be able to render the geometry surface.
- Switch to XNA project Primitives3DWindows.