annotate23d: Human-friendly 3D modelling

William Brown (whb2107)

May 8, 2012

1 Modelling Primitives

Primitives are split into two types: cylinderoids and ellipsoids. Cylinderoids have significantly more features than ellipsoids, in part due to the completeness of their implementation, and in part because they are more powerful primitives.

1.1 Cylinderoids

Cylinderoids are created by drawing a spine on the screen. This spine is made up of every point at which a touch was detected during the swipe gesture – usually, this means that there is no more than about one pixel between each point. This overdensity of the spine is unnecessary, so, on constructing the cylinderoid, the spine is resampled such that the spine points are no closer than 30 pixels away from each other. If resampling would cause the spine to have fewer than 3 control points, that target distance is halved and it attempts to resample again. This is repeated until a good resampling is found.

All cylinderoids are defined by their spine points, their radius at each spine point, and the tilt at each spine point. However, there are a number of annotations that can be applied to each cylinderoid that effect all of these parameters (location, radius and spine). These are stored in fields in the Cylinderoid class. Then, when the mesh is computed, the class computes the locations, radii and tilts with all of the constraints satisfied, and uses those to generate the mesh.

Tilts are slightly different from the other two parameters because they are not guaranteed to be defined everywhere over the spine. While each spine point has a location and a radius, it's possible that the user has defined the tilt in some places, or nowhere at all. `NaN` is stored in the tilt vector where the user has not specified a tilt; the app then linearly interpolates between tilts along the spine, or, if no tilt is assigned anywhere, it sets the tilt to zero along the whole spine.

The mesh of a cylinderoid is generated by sweeping a segmented circular ring along the spine, and connecting each ring with triangles. Each ring is

defined by a location (`spinePoint` in `Cylinderoid::generateMesh`), a tangent vector (mistakenly named `derivative`) and a radius that is perpendicular to the tangent of the spine (`radius`).

1.2 Ellipsoids

Ellipsoids are significantly less advanced than cylinderoids, because they do not support any annotations and have significantly fewer degrees of freedom. Each ellipsoid has a center of mass `com`, a major axis length `a`, a minor axis length `b` and an angle between the major axis and the x-axis, `phi`. The boundary of the ellipsoid in two dimensions in then defined as

$$b = c + (\cos t)a \begin{pmatrix} \cos\phi \\ \sin\phi \end{pmatrix} + (\sin t)b \begin{pmatrix} \sin\phi \\ -\cos\phi \end{pmatrix}$$

Where b is a boundary point, c is the center of mass and t varies from 0 to 2π. The mesh of an ellipsoid is constructed using the same method as that of the cylinderoids, but the radius of each ring is determined using a and b instead of the user-provided radii.

1.3 Annotations

Six annotations are available to the user: same-length, same-scale, same-tilt, connection, alignment and mirroring. The annotation objects themselves, defined in `Annotations.h`, only provide methods for calculating offsets and the like; they do not generate meshes. The Cylinderoid object itself uses the information in the annotations to generate its mesh.

1.3.1 Same-length

Same length annotations ask the user to pick two cylinderoids, and then sets the length of those two cylinderoids equal to the mean of their original lengths. The transformation is taken by translating the cylinder's center of mass to the origin, scaling the spine by the required amount, and then translating back to its original position. Note that the radii are uneffected, so this is a scaling of the spine, not the shape as a whole.

1.3.2 Same-scale

Same scale annotations ask the user to pick two spine points (which can optionally be on the same cylinderoid) and then sets the radii at those spine points to be the mean of the original radii. However, this can often result in strange-looking discontinuities in the radii, so a weak smoothing operator is applied to the radii before drawing.

1.3.3 Same-tilt

Same tilt annotations ask the user to pick two spine points with associated tilt and then sets the tilt at those spine points to be the mean of the original tilts. As the tilt is interpolated over the spine and, in most situations, is very sparsely defined over the spine, no smoothing is necessary here.

1.3.4 Connection

Connection annotations take two shapes – one stationary, and one connector – and an intersection point. The meshes for the two shapes are then generated using all annotations and those meshes are intersected with a ray that is emitted orthogonally from the image plane. If the intersection with the stationary object is z_s from the image plane, and the connector intersection is z_c away, the connector is translated $z_s - z_c$ units along the z-axis, where the z-axis points into the screen.

1.3.5 Alignment

Alignment annotations take one shape which must be the connector in a connection annotation. They find the intersection point between the ray and the connector, and then translate that point to be on the symmetry sheet of the stationary object.

1.3.6 Mirror

Mirror annotations take one shape which also must be the connector in a connection annotation, and they create a duplicate mesh reflected about the symmetry plane of the stationary object. In order to do this, the full mesh is generated for the connector. We then copy the mesh, subtracting two times the projection of each vertex onto the symmetry plane normal from each vertex. This is then unioned with the global mesh to produce a new global mesh.

2 App architecture

2.1 User interaction

The parent file of the entire app is `WorkspaceViewController`, which is responsible for initializing all other parts of the app, as well as the layout of the main view. The layout is defined using a NIB file.

`WorkspaceViewController` also manages what tool is currently active, and acts as a delegate for all touch events. It receives events via the `UIResponder` protocol (`touchesBegan, touchesMoved, touchesEnded`) and either delegates

to `WorkspaceUIView` or `DrawPreviewUIView` or uses them to manipulate the view (zoom or pan).

`DrawPreviewUIView` is used when creating new cylinderoids or ellipsoids. It is reponsible for reading all touch positions and recording them, and when the touch finishes it passes an array of points back up to `WorkspaceViewController`, which creates the new primitive and stores it in the workspace.

The workspace is a `WorkspaceUIView`, and it's responsible for most of the interesting interaction between user and app. It keeps track of what shapes exist in the workspace, and also handles all touch events related to annotations or manipulating existing shapes. All of the stories for creating new annotations are defined here, in methods whose names match the annotation name. It also draws unselected shapes.

If at any time a shape is selected, the workspace stores a reference to a `ShapeTransformer` for that shape. Shape transformers are responsible for all of the handles associated with an object. They are also responsible for drawing selected objects – this includes handles and annotations. Most annotations are handled by calling class methods in `AnnotationArtist`.

The `EllipsoidTransformer` class is very simple; it handles translate-rotate-scale (TRS) for ellipsoids, and handles manipulation of the major and minor axis handles. These handles have a simple interaction model; as the user moves the handle, the transformer finds the projection of the user's touch on the axis which they are manipulating, and sets that dimension to that projection.

The `CylinderoidTransformer` class is much more complex because of the increased complexity of cylinderoids. It handles TRS for the shape as a whole, but the user can also select a spine handle and move that handle, or pinch to resize the radius at that point, or tap again and drag to change the tilt at that point. All of this is handled in `CylinderoidTransformer`.

2.2 Mesh generation

When the user requests a render or exports the model to an OBJ file, `MeshGenerator::globalMesh` is invoked on the workspace. This method requests meshes from all of the primitives defined in the workspace, and then uses `Mesh::combine` to combine all of these meshes into one single mesh.

The mesh format is unfortunately complex, but it is what's required by OpenGL ES due to its lack of immediate mode. Each mesh object is an array of floats, where each vertex is defined as six numbers: the x, y z of its location, and the x, y z components of the normal for that vertex. Each triangle is a block of three of these vertex definitions; each triangle is therefore a block of 18 floats which map to three verteces as described above.

This mesh is then passed to `GlkRenderViewController`, which creates a modal popover containing an OpenGL view, compiles the vertex and fragment shaders and draws the mesh to the view.

Wavefront OBJ file generation is handled by `ObjExporter`, which uses the global mesh from `MeshGenerator` to create a file in the iPad's local storage that contains an OBJ file. Most of the complexity in this file comes from the need to merge coincident verteces; since the mesh data structure doesn't have any adjacency information, a naively-generated mesh file would be a bag of triangles without adjacency. This would be useless for most graphics algorithms and would result in significant wasted storage space. Verteces are merged using their position to compute adjacency.

3 Significant completed features

- Creation and manipulation of cylinderoids and ellipsoids in 2D, including least-squares fitting of drawn ellipses
- Same-length, same-scale, same-tilt, connection, alignment and mirror annotations
- 3D mesh generation of all primitives, taking into account all annotations
- OBJ file exporting via email
- Background images using iOS's built-in photo browser
- Interactive tutorial that informs the user what results their actions will have

4 Future work

A list of things that were not completed that were discussed, or that I wish had been done:

- Connections on ellipsoids. This is easy enough, and much of the infrastructure is already there. It was not completed due to lack of time.
- Manipulation of the OpenGL view and OpenGL preview picture-in-picture.
- Cross-section shape adjustment
- Better UI for manipulating annotations
- Symmetry sheet manipulation