Spring (spring.*) 2.5 punts

Write **VS+FS** to simulate the cyclic expansion/compression of the 3D model as if it was a spring (see attached **spring.mp4**).

The VS will compute the animation that will consist of two phases **that will be repeated every 3.5 seconds.**

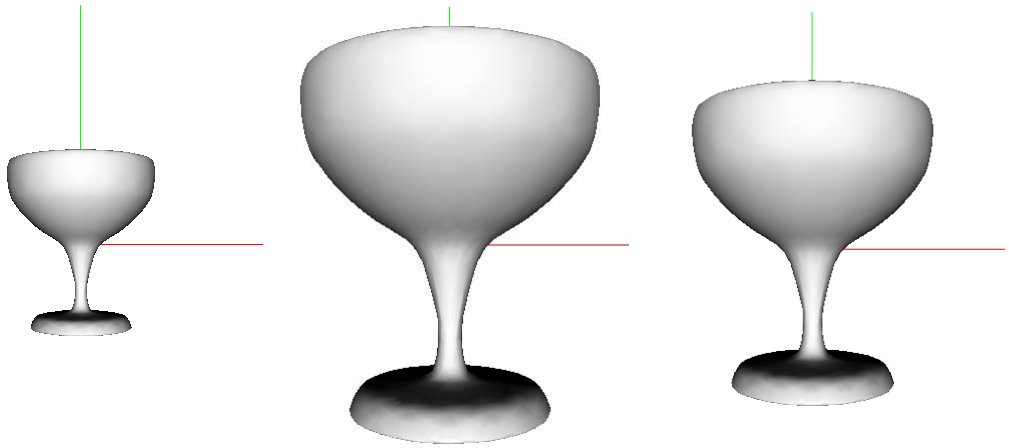
The first phase (expansion) will last **0.5 seconds** and will move vertices from the origin to its original position in *model space*. As linear interpolation parameter between the two positions above, use the value $(t/0.5)^3$, where t is the time in seconds from the start of the period (for example, when time = 4, t = 0.5).

The second phase (compression) will last **3 seconds**, and will move the vertices from its initial position in *model space* towards the origin. However, in this phase we wish vertices to move at uniform speed. You should figure out how to compute the linear interpolation parameter, from a t value that within each period should be in [0.5, 3.5).

After moving the vertex in model space, we should transform it onto *clip space,* as usual. The color will be the gray defined by the Z components of the normal in *eye space*.

The FS will do default tasks.

Results at times 0.4, 0.5, 1:



Files and identifiers:

`spring.vert, spring.frag`

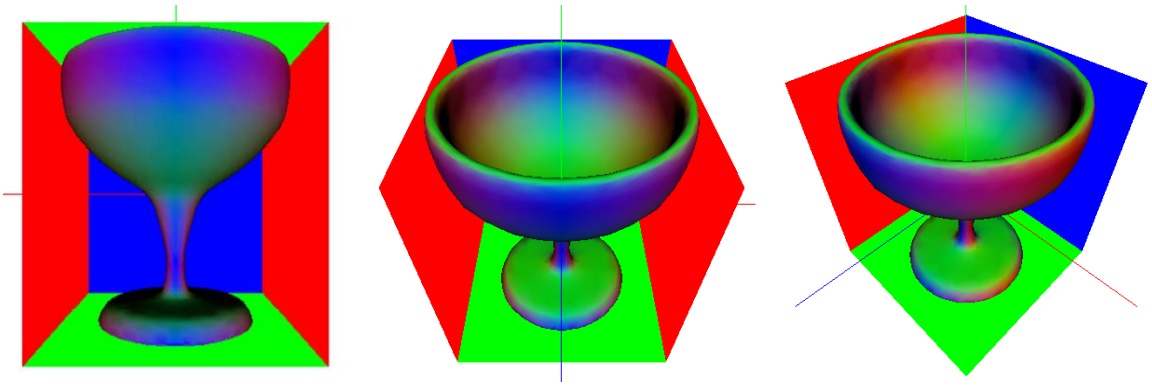
Walls (walls.*) 2.5 punts

Write **VS+GS+FS** to draw each triangle of the 3D mode as usual, and also the faces of the scene bounding box (boundingBoxMin, boundingBoxMax).

The VS will do the usual tasks.

The GS will draw each triangle as usual. If **gl_PrimitiveIDIn** is 0, the GS will also draw the bounding box faces. Their color will depend on orientation: red for faces perpendicular to X axis, and green/blue for faces perpendicular to Y/Z axis.

These faces should be drawn only if they are *backface* with respect to the camera, so that the bounding box will never occlude the model:



For the backface test, you could use the sign of P·N (where P is an arbitrary vertex of the face, and N the face normal, both in eye space). Alternatively, you could check the sign of the distance between the camera position and the (a,b,c,d) plane of the face.

The **FS** will just write the color from the VS.

Files and identifiers:

`walls.vert, walls.geom, walls.frag`

Equi (equi.*) 2.5 punts

Write **VS+FS** to render panoramic views encoded with an equirectangular projection. Use **verandah.png** (Greg 'Groggy' Lehey) from **/assig/grau-g/Textures/** as texture:

In this texture, a texel (s,t) represents the color along the direction of the unit vector (x,y,z) given by:

$\theta = 2\pi s$
$\psi = \pi(t-0.5)$

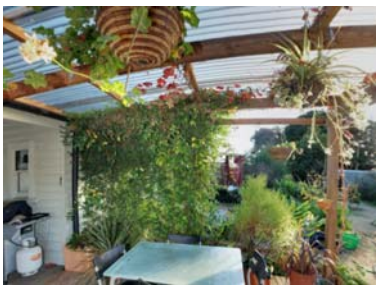
$x = \cos(\psi)\cos(\theta)$
$y = \sin(\psi)$
$z = \cos(\psi)\sin(\theta)$

These shaders should be tested with the sphere model (sphere.obj), using a camera inside the sphere.

VS: besides writing gl_Position, will also provide the FS the **object space** coordinates of the vertex.

FS: using the (x,y,z) values from the VS, the FS will compute the (s,t) corresponding to direction (x,y,z). For this, you should first compute angles θ, ψ from (x,y,z) (isolating them in the equations above); you will need **asin**, **atan** GLSL functions), and then compute (s,t) from θ, ψ. The fragment color will be that of the texture at (s,t). The GLSL **atan**(a, b) function computes the inverse tangent of a/b, returning a value in $[-\pi, \pi]$.

Results (sphere.obj, camera near the origin):


Files and identifiers:

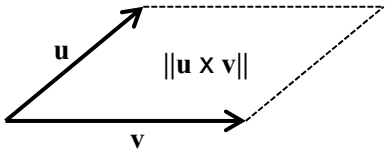
```
equi.vert, equi.frag
uniform sampler2D panorama;
const float PI = 3.141592;
```

Base (base.*) 2.5 punts

Write a **plugin** that outputs to standard output (use cout), the area of the bottom half of the model (assume the scene has a single object).

You should implement only **onPluginLoad**(). This method will have to traverse the faces of the object, and compute the area as the sum of the area of its faces. You should only accumulate the area of those faces whose minimum Y is below the Y of the object bounding box center (use **Object::boundingBox()** method).

All models will be triangle meshes. The area of a triangle can be computed taking into account that the length of the cross product of two vectors **u**, **v** is the area of the parallelogram defined by **u, v**:



The plugin will write the area of the bottom part of the object, preceded by the literal “Area:”

Example (default object):

Area: 18.6827

Files and identifiers:

`base.pro, base.h, base.cpp`