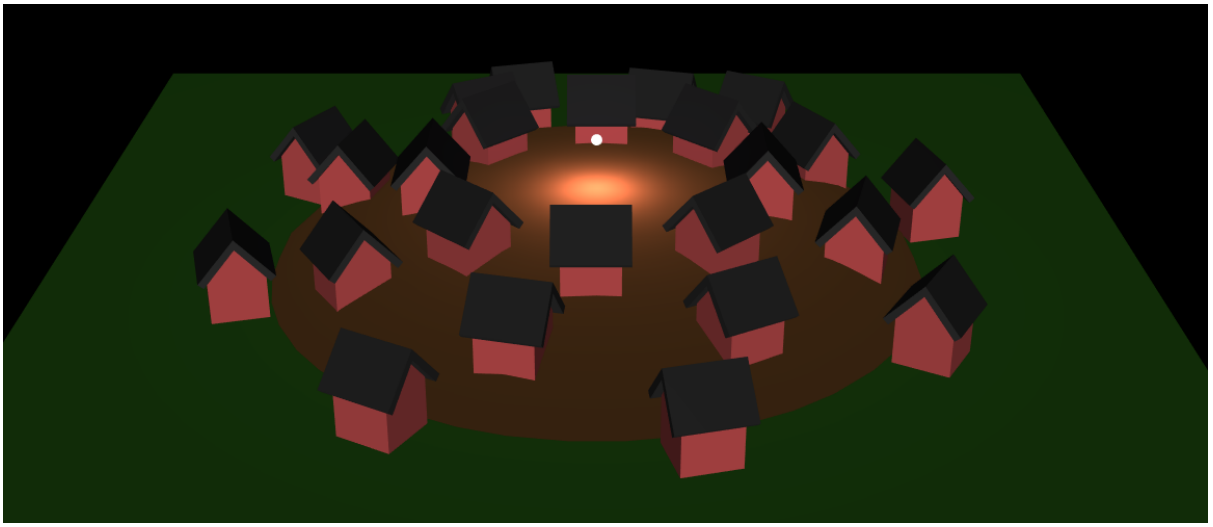
Exercise 5

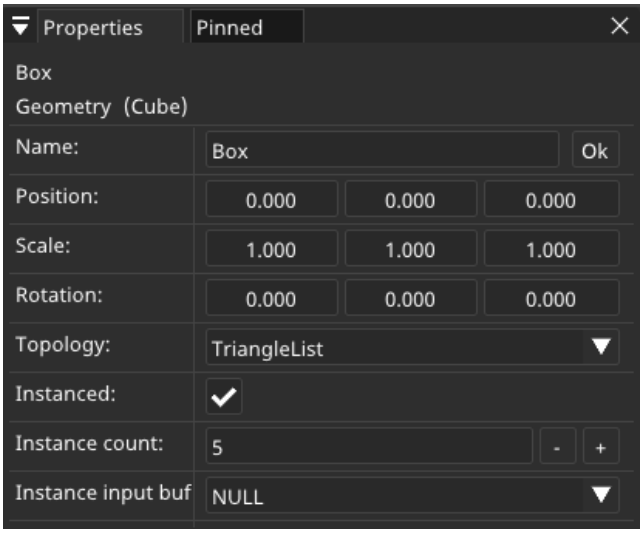
Computer Graphics - COMP.CE.430

Transforms and Instanced Rendering

This week's goal is to get familiar with working with transformation matrices and learn how we can use matrices to render multiple instances of the same triangle mesh with different transforms. This technique is commonly called *instancing* or *instanced rendering*. The image below illustrates the concept of instancing. By using instancing, we can avoid having to duplicate the triangle mesh.



You can use instanced rendering in SHADERed by going to the object's *Properties* tab, checking the *Instanced* checkbox, and setting *Instance count* to some number. You can query the current instance ID in the vertex shader using the built-in variable *gl_InstanceID*. Note: You may want to create a new shader pass for the objects that you want to instance. This way you can avoid instancing every object in your scene.



Problem set

1. Define helper functions for creating transformation matrices that do rotation, scaling, and translation. The functions should all return a mat4. You should end up with 5 different helper functions; one for each axis of rotation, one for translation, and one for scaling. **(1.5p)**

   The definitions for the different transformation matrices from the lecture slides are repeated here:

   $$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_y = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

   where θ is the angle of rotation around the axis.

   $$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

   where t_x, t_y and t_z denote how much the object is moved along each coordinate axis.

   $$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

   where s_x, s_y and s_z denote how much the object is scaled along each axis.

   Refer to the OpenGL wiki for details on how to initialize and index matrices.

2. Use instanced rendering to render the same 3D object multiple times with different transforms. **(1p)**

   Check the *Instanced* check box in the properties tab of the 3D object and set the instance count to a number of your choice. Use the built-in variable *gl_InstanceID* to create different transformation matrices for each instance. See if you can put all of the instances in some regular pattern such as a grid or a circle.

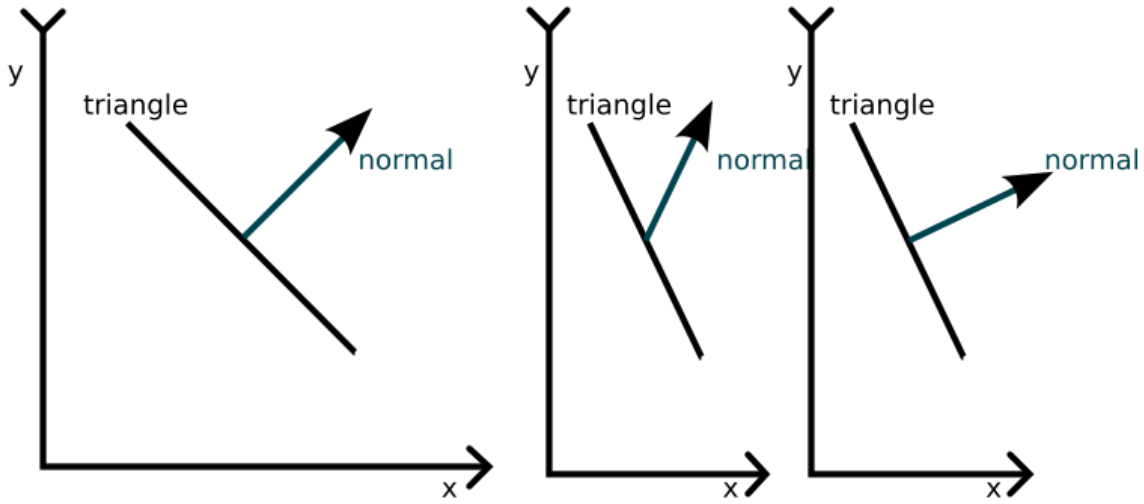
   Remember that you need to multiply the input vertex position by the transformation matrix *before* multiplying it by the *ViewProjection* matrix.

3. Transform the normal vectors correctly using the transformation matrix and use it to compute lighting for a transformed 3D object. **(0.5p)**

For shading to work correctly, you need to also transform the normal vector. Remember from the lectures that you can convert *directions* to homogeneous coordinates by setting the *w* component to zero. Use the correctly transformed normal vector to compute *Phong* or *Blinn-Phong* shading from the previous exercise.

If your transformation matrix contains only translation, rotation and *uniform* scaling, you can use the same matrix to also transform the normals. This is the most common case in real-time computer graphics and it works most of the time.

The situation where this breaks down is if your transformation matrix has *non-uniform* scaling. The following image illustrates what happens:



The figure in the middle shows the result after scaling the x axis by 0.5. The rightmost figure shows the correct normal vector.

In general, to correctly transform normals, you need to multiply them by the **transpose of the inverse** of the transformation matrix that you used to transform the position. You can use the inverse and transpose functions in GLSL to achieve this. You only need to do this if you used non-uniform scaling in your transformations; translations don't affect normal vectors, a series of rotations only rotates normal vectors and *uniform* scaling only changes their length but not their direction.

Remember to also normalize the normal vector after transforming it and before using it in shading equations. Transformations that include scaling will cause the normal vector to no longer be of unit length. The interpolation that the graphics pipeline does

to the variables that are passed from the vertex shader to the fragment shader also changes the length of the normal vector.

Bonus

1. Create your own projection matrix and replace the built-in uniform with it. **(1p)**

   You'll need to know the aspect ratio to create a projection matrix. You can compute it by using the SHADERed built-in *ViewportSize* uniform variable. The *ViewportSize* uniform is a vec2.

   The definition of a reverse Z projection matrix was given during the lecture. Unfortunately, we can't use the reverse Z version in SHADERed (we would need to be able to change the clear value of the depth buffer and SHADERed doesn't expose that option). We'll need to use a standard projection matrix instead. The definition for a standard projection matrix usable with OpenGL conventions is

$$P = \begin{bmatrix} \frac{1}{a\tan(\alpha/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\alpha/2)} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

   where a is the aspect ratio, α is the field of view, f is the distance to the far plane and n is the distance to the near plane. The field of view should be in radians. Reasonable values for the near and far plane distance are something like 0.1 for the near plane and 1000.0 for the far plane.

   Replace the *ViewProjection* matrix in the vertex shader with the product of the built-in *View* matrix and your own projection matrix.

2. Create a view matrix in the vertex shader and use that instead of the SHADERed built-in uniform matrix. **(1p)**

   You can create a view matrix by creating a model matrix for the virtual camera and taking the inverse of that. Remember that the three leftmost columns of a rotation matrix define the coordinate axes of the object in the reference space.

   To create the basis vectors for the camera, you can define a forward direction vector and compute the other vectors based on that using cross products. The view space *Z* axis in the OpenGL convention points *behind* the camera, so the *Z* basis vector is the negation of the forward vector. If you want the camera to point at the origin, you can use the camera position uniform and normalize it to define the *Z* basis vector.

To compute the *X* vector, you can compute it by taking the cross product of a predefined up vector (the vector (0, 1, 0) in our case) and the *Z* vector. The cross product operation takes in two vectors and outputs a vector that is perpendicular to both of the input vectors and its direction follows the right-hand rule. The output vector is not guaranteed to be unit length, so you need to normalize it.
You can use the built-in cross function in GLSL as follows:

```
vec3 X = normalize(cross(up, Z));
```

You can compute the *Y* vector in a similar manner using the *Z* and *X* vectors:

```
vec3 Y = cross(Z, X);
```

The *Y* vector does not need to be normalized here, because the cross product between two orthonormal vectors produces a unit length vector.

Create a 4x4 matrix using the basis vectors for the rotation component and the camera position for the translation component. This kind of matrix is often called a “LookAt matrix” because we defined it in terms of where we wanted the camera to look at. The view matrix is the inverse of the camera’s “LookAt” matrix.

Replace the built-in *View* matrix in the vertex shader with your own view matrix.