**Tutorial 3: Animation**
Concatenate rigid body transforms to move objects or the camera.

- Home

In this tutorial we are going to make things move, or animate. The goals are to be able to move the camera to look at things from different angles, and to make a part of the scene move as time progresses, and even interact with user input.

We will focus on transforming geometry so we abstract away the setup of vertex data and textures. We will import a 3D model from an .OBJ file (wavefront) instead of specifying the data manually.

Run the program, press mouse buttons and move the mouse around and see what happens. Then look through the code and make sure you understand it all so far.

**Task 1: Moving the car**
In this lab we finally move completely into the 3d space. That is, we will have a Model-, a View- and a Projection matrix which will be used to transform all vertices from their model-space coordinates to their window coordinates.

In OpenGL and, by extension, in the GLM library we use to handle math types, matrices are stored in column-major order. That way, to create the matrix

$$M = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

One would write

```
mat4 M(a, e, i, m,
       b, f, j, n,
       c, g, k, o,
       d, h, l, p);
```

The first matrix we focus on is the model matrix, which tranforms points from the model space (specific to a model) to the world space (common for all models).
As of now, the model matrices for the city and the car are both the identity matrix, which means that the vertex coordinates will have the same position in world space as in model space, so we will start by applying a translation to the car (but keeping the city fixed).
Replace the modelMatrix of the car with a translation matrix that you construct yourself. A translation matrix looks like a 4 by 4 identity matrix, but the last column contains the translation. When we apply the translation matrix $T$ on a vertex position $p$ we actually make this computation:

$$Tp = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

Taking this last expression, in code we can modify the existing identity matrix to add a translation to it in the following way:

```
T[3] = vec4(tx, ty, tz, 1.0f);
```

To test this out, first make a hardcoded translation, e.g. by $t = (0, 0, 5)$, and make sure that the car moves while the city stays the same.
Then add keyboard controls for the translation, e.g. the arrow keys. One way is to add 'speed' translation to the $z$-component for each frame that Up is pressed down (try speed 0.3f for this scene).
Now add controls for translation along the $x$-$z$ plane and try it out:

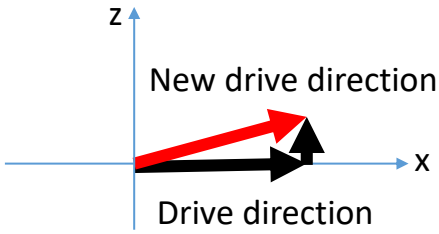
```
// check keyboard state (which keys are still pressed)
const uint8_t *state = SDL_GetKeyboardState(nullptr);

// implement controls based on key states
const float speed = 10.f;
if (state[SDL_SCANCODE_UP]) {
        T[3] += speed * deltaTime * vec4(0.0f, 0.0f, 1.0f, 0.0f);
}
if (state[SDL_SCANCODE_DOWN]) {
        T[3] -= speed * deltaTime * vec4(0.0f, 0.0f, 1.0f, 0.0f);
}
if (state[SDL_SCANCODE_LEFT]) {
        T[3] += speed * deltaTime * vec4(1.0f, 0.0f, 0.0f, 0.0f);
}
if (state[SDL_SCANCODE_RIGHT]) {
        T[3] -= speed * deltaTime * vec4(1.0f, 0.0f, 0.0f, 0.0f);
}
```

And use $T$ as the model matrix for the car:

```
carModelMatrix = T;
```

**Task 2: Steering**
Next up, we will add some steering to the car by rotating its orientation. A pure rotation around the origin (no scaling, no translation) is just a change of base and the new base vectors are the columns of the upper left 3 by 3 matrix. Pure rotations should be an orthonormal base (each base vector should have unit length, and all base vectors should be pair-wise orthogonal).




$$R = \begin{bmatrix} r_{00} & r_{01} & r_{02} & 0 \\ r_{10} & r_{11} & r_{12} & 0 \\ r_{20} & r_{21} & r_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From start, we will use an identity matrix for rotation matrix, and we will replace the side-way translation when we press Left/Right for a rotation of the car. We can perform the rotation in terms of an angle, but we will make it quick and dirty for now. We will keep $y$ and only rotate in the $x$-$z$ plane. Add/subtract a portion of the third base vector to the first base vector with Left/Right:

```
const float rotateSpeed = 2.f;
if (state[SDL_SCANCODE_LEFT]) {
        R[0] -= rotateSpeed * deltaTime * R[2];
}
if (state[SDL_SCANCODE_RIGHT]) {
        R[0] += rotateSpeed * deltaTime * R[2];
}
```

And then we make $R$ orthonormal again by normalizing $x$ and recomputing $z$:

$$R_0 = \frac{R_0}{||R_0||}$$

$$R_2 = R_0 \times R_1$$

Which becomes:

```
// Make R orthonormal again
R[0] = normalize(R[0]);
R[2] = vec4(cross(vec3(R[0]), vec3(R[1])),  0.0f)
```

Try $R$ for model matrix:

```
carModelMatrix = R;
```

Then try the concatenation of $T$ and $R$ for model matrix:

```
carModelMatrix = T * R;
```

After steering (rotating) the car, we still are 'driving' in the same direction. Now make the translation happen in the direction of the car.

**Hint:** You need to apply the car's current rotation to the untransformed velocity vector, so that the next translation will follow that new direction. The untransformed velocity vector should be the same as the original forward direction of the car.

What does the model matrix transform? (strictly speaking)

> It transforms the coordinate-matrix of the model based on rotation and translation.

Will $RT$ produce the same transformation as $TR$? Why/Why not?

> No, reversing order of the cross multiplication reverses the direction of the product.

**Task 3: Time dependent animations**
Render the car another time (a second `carModel->render()` call), with another model matrix, effectively reusing the car model and adding another car to our application. Make the model matrix of the second car such that it drives in a circle around your starting position. The position in the track should depend on the float variable `currentTime`. For this task, use glm's functions for transformations.

Translate:

```
mat4 translate(vec3 t);
```

Rotate (around an axis):

```
mat4 rotate(float radians, vec3 axis);
```

And if you like, the scale matrix (one scale factor per axis):

```
mat4 scale(vec3 scale);
```

**Task 4: Adding camera control**
The camera is defined by the two 3D vectors `cameraPosition` and `cameraDirection` that are declared as global variables. We will control these with the mouse and keyboard. We rotate the camera direction based on the mouse motion when the left mouse botton is pressed down. With a horizontal mouse movement, we will rotate the camera direction around the `worldUp` direction, and with a vertical mouse movement, we will rotate around an axis orthogonal to both the `worldUp` and `cameraDirection`. Replace the code for the mouse motion event:

```
if (event.button.button & SDL_BUTTON(SDL_BUTTON_LEFT)) {
        float rotationSpeed = 0.005f;
        mat4 yaw = rotate(rotationSpeed * -delta_x, worldUp);
        mat4 pitch = rotate(rotationSpeed * -delta_y, normalize(cross(cameraDirection, worldUp)));
        cameraDirection = vec3(pitch * yaw * vec4(cameraDirection, 0.0f));
}
```

For translation, we will use the W,S keys. Your task is to make the `cameraPosition` move a small amount in the `cameraDirection` when W is pressed, and in the opposite direction when S is pressed.

We will now replace the constant view matrix (in display) with one we control with our keyboard and mouse. The view matrix transform world space coordinates into view space coordinates. In view space, the cameras position is at the origin. So the first transform (the right-most) is a translation that puts the cameras world space position as the origin:

```
mat4 viewMatrix = cameraRotation * translate(-cameraPosition);
```

In view space, we look down the negative z-axis (at least in OpenGL). So we need to make a rotation matrix that rotates the world space coordinate (after the translation). We also need to decide what should be the up direction of the camera, cameraUp. We can choose the direction that's closest to the worldUp direction and orthogonal to the cameraDirection. The third base vector is then fixed since it has to be an orthonormal base. The desired base vectors are [cameraRight, cameraUp, -cameraDirection] and we compute the base vectors as:

```
// use camera direction as -z axis and compute the x (cameraRight) and y (cameraUp) base vectors
vec3 cameraRight = normalize(cross(cameraDirection, worldUp));
vec3 cameraUp = normalize(cross(cameraRight, cameraDirection));

mat3 cameraBaseVectorsWorldSpace(cameraRight, cameraUp, -cameraDirection);
```

To get the rotation matrix that rotate the world into this base, we take the inverse of the matrix. Since the rotation is an orthonormal base, the inverse is just the transpose. We can now put together the final view matrix:

$$\mathrm{CameraBaseVectors}_{\mathrm{ws}} = \begin{bmatrix} \mathrm{cameraRight}_x & \mathrm{cameraUp}_x & -\mathrm{cameraDir}_x \\ \mathrm{cameraRight}_y & \mathrm{cameraUp}_y & -\mathrm{cameraDir}_y \\ \mathrm{cameraRight}_z & \mathrm{cameraUp}_z & -\mathrm{cameraDir}_z \end{bmatrix}_{\mathrm{ws}}$$

$$R_{\mathrm{camera}} = {\mathrm{CameraBaseVectors}_{\mathrm{ws}}}^{-1} = \begin{bmatrix} \mathrm{cameraRight}_{\mathrm{x}} & \mathrm{cameraRight}_{\mathrm{y}} & \mathrm{cameraRight}_{\mathrm{z}} & 0 \\ \mathrm{cameraUp}_{\mathrm{x}} & \mathrm{cameraUp}_{\mathrm{y}} & \mathrm{cameraUp}_{\mathrm{z}} & 0 \\ -\mathrm{cameraDir}_{\mathrm{x}} & -\mathrm{cameraDir}_{\mathrm{y}} & -\mathrm{cameraDir}_{\mathrm{z}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{\mathrm{view}} = R_{\mathrm{camera}} T_{\mathrm{camera}}$$

```
mat4 cameraRotation = mat4(transpose(cameraBaseVectorsWorldSpace));
mat4 viewMatrix = cameraRotation * translate(-cameraPosition);
```

Move around in the world and see if the camera controls are working properly.

**Task 5: Perspective Transform and Z-Fighting**
The third component of the modelViewProjection transform is the projection transform. We use the glm function `perspective()` to generate this matrix. Have a look at the arguments to this function and play around with them. You can do it interactively with the gui, which you can toggle by pressing G (you can also comment out the `if` surrounding the call to `gui()` in `main()`).

Now uncomment the `drawGround()` call in the `display()` function. Now you will see that in the middle of the scene there's a new flat square. You will also notice that it looks like it's broken, and when you rotate the camera it will look like the brokenness changes. This is what we call z-fighting.

Play around a bit with the values in the gui and try to find out which setting helps reduce or remove this effect.

What parameter did you change to reduce the z-fighting effect at the default camera distance?

Near Plane

What value did you set it to?

0.82

**Task 6: [OPTIONAL] Add your own camera control**
Think about how games control the camera with W,A,S,D (or arrow keys) and the mouse. Try to implement something similar.

Suggestions: First person while driving the car. Third person while driving the car.

---

**When done, show your result to one of the assistants. Have the finished program running and be prepared to explain what you have done.**

---