Homework 2 - Anna Carini

Objects:

Every element of the scene is a cube, arranged in a hierarchical structure. I've added three components for the tail (*tail1*, *tail2*, *tail3*) so that *tail1* is a "sibling" of *rightUpperLeg*, *tail2* is a "child" of *tail1*, *tail3* is a "child" of *tail2*.

To handle the animation, I have also created an array called *theta*, which contains: 17 values for the angles of the joints, 3 values for the x,y,z translation of the cat, 1 value to vertically translate the legs (to make the sitting position appear more natural), 1 vertical scale factor for the "stretch and squash" of the jump animation.
The values contained in *theta* are used, inside the function *initNodes()*, to compute the transformation matrices *m* of the nodes.

Animation:

The animation is created as linear interpolation of keyframes.

This is done by defining the array *keyFrame*, which contains all the keyframes. Every element of the array *keyFrame* (i.e. every keyframe) is an array of values, of the same size as *theta*, containing the angles/displacements/etc.

I have also defined a value: *framesBetweenKeyFrames*, which is the quantity of interpolated frames between each couple of consecutive keyframes. This variable is initialized as 10, but can be changed through the "animation speed" slider to make the animation faster/slower.

The animation is then handled inside the *render()* function. If the animation is active, the function interpolates the current keyframe with the next one, and assigns the result to the array *theta*. Then the nodes corresponding the cat's body parts (nodes 0 to 13) are re-initialized, in order to update their tranformation matrices.

Textures:

To be able to use at the same time a color texture and a bump texture, I have added to the first fragment shader two uniform sampler2D: *uTexture* and *uBump*.

Inside the *init()* function of the javascript file, I load the images, I turn them into textures using the function *configureTexture()*, and I add them to an arrray called *textures*. I then associate the texture unit 0 to *uTexture* and the texture unit 1 to *uBump*.

I also use two boolean uniform variables in the fragment shader: *uTextureActive* and *uBumpActive*, which are used to determine whether to use the textures or not.

Every time I draw a component which uses textures, first of all I send the value "true" to *uTextureActive* and/or *uBumpActive*, then I activate the texture units (*gl.activeTexture(...)*) and bind the correct textures (*gl.bindTexture(...)*).

Color texture:

The textures are applied in the first fragment shader. If *uTextureActive* is true, *materialDiffuse* and *materialAmbient* are modified in this way:

```
vec4 texColor = texture(uTexture, vTexCoord);
materialDiffuse = (texColor.a)*texColor + (1.0 - texColor.a)*materialDiffuse;
materialAmbient = materialDiffuse;
```

Where the second line assigns to *materialDiffuse* the color of the texture in a way that is proportional to its opacity: this is done to correctly handle transparent textures.

By modifying the properties of the material in this way, instead of directly assigning the color of the texture to *fColor*, the lights and shadows are still computed correctly.

Bump texture:

If *uBumpActive* is true, the normal *N* of the fragment is computed in this way:

```
vec4 NN = texture(uBump, vTexCoord);
mat3 M = mat3(B.x, B.y, B.z,
              T.x, T.y, T.z,
              N.x, N.y, N.z);
N = (2.0*NN.xyz - 1.0);
N = M * N;
N = normalize(N);
```

The reasoning behind this code is the following: the goal is to assign to the normal *N* the value given by the bump texture.

But the bump maps I have are mostly blue, because they are meant to be put on a surface whose normal vector is the positive Z axis, i.e. (0,0,1), which in the RGB space corresponds to the color blue.

For the same reasoning, the color green (0,1,0) in the bump map corresponds to the tangent vector (positive Y axis) and the color red (1,0,0) corresponds to the binormal vector (positive X axis).

So to apply the bump map to a surface that is not oriented in this way, I need to modify the color of the bump texture, mapping the "blue" of the texture to the normal vector, the "green" to the tangent vector, the "red" to the binormal vector.

In other words I need to multiply every color of the texture by a matrix M such that:

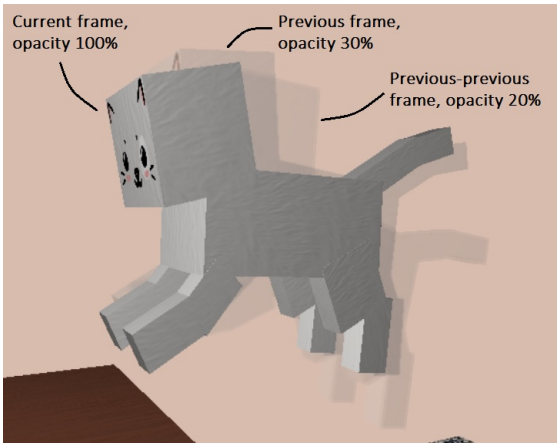
$$\begin{cases} (1,0,0) * M = \text{binormal} \\ (0,1,0) * M = \text{tangent} \\ (0,0,1) * M = \text{normal} \end{cases}$$

From this, it's clear that the matrix must be: $M = (\ \text{binormal}^T \ |\ \text{tangent}^T \ |\ \text{normal}^T\ )$.

Since GLSL matrices are written in a column-major way, the final expression is the one reported in the code above.

The expression N = (2.0*NN.xyz - 1.0) is used to transform colors, whose coordinates range from 0 to 1, into vectors with coordinates that range from -1 to 1.

Motion blur:



The motion blur effect is achieved by rendering three consecutive frames on three textures, and applying them on a rectangle the same size as the canvas.

To be able to render on textures, I have created a *framebuffer* variable, plus four textures to use as color attachments, and three renderbuffers to use as depth attachments.

In the render function, if the motion blur is active, first of all I bind the framebuffer, and I associate to it the first texture (*texture2*) and the first renderbuffer (*renderbuffer2*). Then I draw the first of the three frames by setting *theta* equal to the array *prevPrevFrame*, initializing the nodes, and drawing them with *traverse()*. This draws the scene inside *texture2*.

I use a similar process to draw the second frame on *texture3* and the third frame on *texture4*. On *texture5*, instead, I render the third frame again but only drawing the cat (I'll explain why later).

Then, I unbind the framebuffer, and I can finally start rendering on screen.

To render on screen, I use the second program to draw a rectangle with x-y clip coordinates equal to (-1,-1), (-1,1), (1,1), (1,-1). The second vertex shader is simply a "pass through" shader, so that the rectangle fills the whole canvas.

The second fragment shader takes the three textures on which the frames have been drawn, blends them and applies them as texture on the rectangle.

I chose to blur the cat in *texture2* and *texture3*, to make the "current frame" *texture4* appear more sharp by contrast. To avoid blurring the carpet and the table, the blur is only applied to the fragments which have a different color in *texture4* with respect to *texture2* and *texture3* (since the cat is the only part that changes from one frame to the other).

The equivalent in texture coordinates of the size of a pixel is obtained through two uniform variables, *onePixelVertical* and *onePixelHorizontal*, which are computed in the javascript file respectively as *1/canvas.height* and *1/canvas.width*.

These values are used to blur the cat of *texture2* by three pixels:

```
float distanceVert2 = 3.0*onePixelVertical;
float distanceHoriz2 = 3.0*onePixelHorizontal;
colorTex2 = (texture( uTexture2, vec2(x+distanceHoriz2, y))
           + texture( uTexture2, vec2(x, y+distanceVert2))
           + texture( uTexture2, vec2(x-distanceHoriz2, y))
           + texture( uTexture2, vec2(x, y-distanceVert2))) / 4.0;
```

And, in a similar way, to blur the cat of *texture3* by one pixel.

To avoid making the current frame transparent, I use only the color given by *texture4* when *texture5* is different from the background color, i.e., when the current fragment is part of the cat in the current frame.

Otherwise, the final color is computed as 50% of the "current frame" (*texture4*), plus 30% of the "previous frame" (*texture3*), plus 20% of the "previous-previous frame" (*texture2*).