M30242 – Graphics and Computer Vision

Lecture 07: Animation

Overview

- Animation and transformation
- Animation loop
- View control and scene navigation
- Object path calculation

Introduction

- We have learnt how to create static scenes in WebGL.
- For many applications, we may want to navigate the scene and move the objects around, as in simulations and games.
- In this lecture, we introduce the principles and techniques to create dynamic scenes and objects.

Animation & Transformations

- The positions of objects and the viewpoint of the virtual camera are controlled by the modelview transformation – the combination of model transformation and the view transformation.
- View transformation controls the position of the viewer, i.e., the camera. If we want to animate camera effects, e.g., zoom in and out, panning, we need to change this transformation.
- Model transformation controls the position and orientation of an object. If we want to have a moving object among a group of static objects, we need to alter the model transformation of the object in real time.

Modify Transformations

- The modelview transformation can be changed using elementary transformations:
  - translation,
  - rotation, and
  - scaling, if necessary (e.g., dealing with object with changing sizes).

- We will see where and how these can be done.

Animation Loop

- For moving objects, the modelview transformations are constantly changing. The changes must be updated in real time.
- In animation, the drawing method is called repeatedly in a drawing (animation) loop and each call of the method re-draws the scene and produces a frame.
- In JavaScript, the animation control loop can be set by
  - `setInterval()` or `setTimeout()` for ***fixed time intervals***, or
  - `requestAnimationFrame()` for ***variable time intervals***.

Fixed Frame Intervals

- For example

```
function draw() {
    // 1. Update the positions of the objects
    // 2. Draw the current frame of the scene
}


function startup() {
    // Do your usual setup and initialization
    setInterval(draw, 16.7);
}
```

- But for rendering of complex scene that requires long and varying frame times, it is advantageous to use `requestAnimationFrame()`.

Flexible Frame Intervals

- A typical use of `requestAnimationFrame()` looks like this:

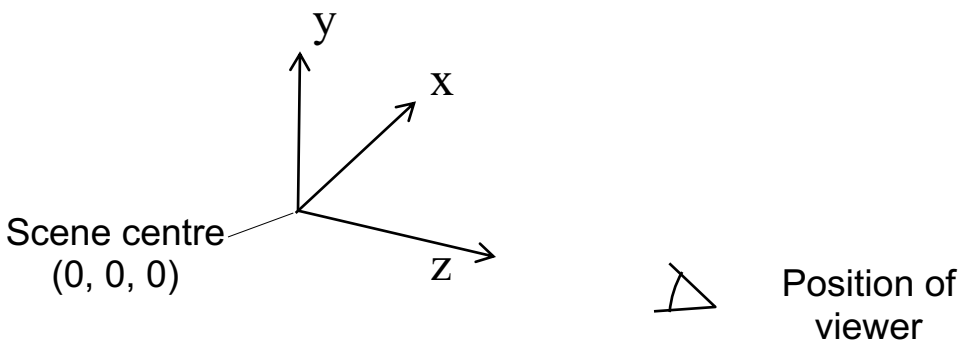
```
draw() {
    For each object {
        //modify model transformation M_model
        //modify view transformation M_view, if necessary.
        //calculate new modelview transformation
          M_modelview = M_model X M_view
        //upload the modelview transformation to the vertex
          shader
        //draw the object
    }
    //make request for the next call of the method draw()
    requestAnimationFrame(draw);
}
```

Viewpoint Control

- As usual, viewer's position is set by calling `mat4.lookAt()` method, which generates a view transformation with the given eye position, focal point, and upward pointing axis:

```
mat4.lookAt(eye, center, up, dest)
```

- Where
  - `eye` – the position of the viewer in the form of a 3D vector, vec3, .
  - `center` – the point the viewer is looking at. In the form of a vec3,
  - `up` – the direction of "up". Also a vec3
  - `dest` – a 4D matrix where the generates transformation is written into.

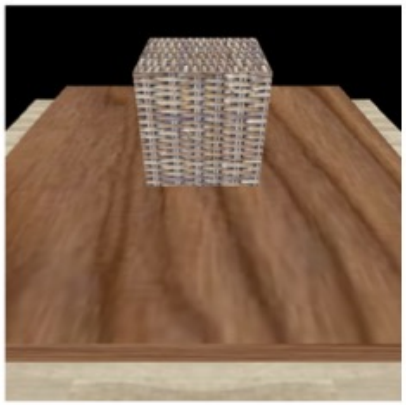


Change View by Setting Camera Location

- The focal point is normally set at the origin (0, 0, 0) of the world coordinate system – the centre of the scene.
- To view the entire scene, the viewer's position (eye vector) need to be set at some distance away from the centre of the scene, e.g., (0, 0, 10), otherwise half of the scene (the positive half of Z axis) will be invisible.

```
mat4.lookAt([0,0,10], [0,0,0], [0,1,0], modelViewMatrix);
```

  where the up direction is set to be upward (0, 1, 0), i.e., y direction.
- Views rendered by using different `eye` vectors:



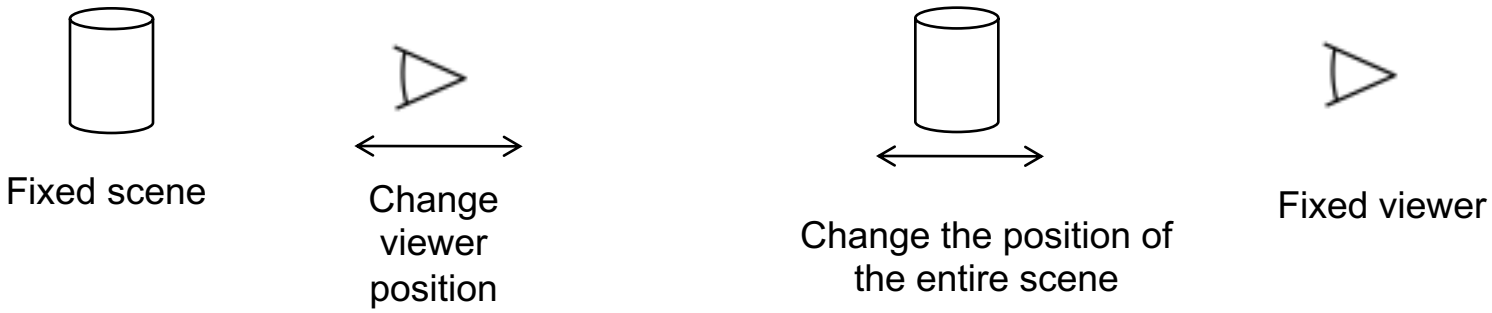
`eye= [0,4,5]`

`eye= [0,4,10]`

`eye= [0,4,20]`

Change View by Changing Model Positions

- Alternatively, viewpoint can be changed by moving the entire scene towards the negative direction of Z axis (i.e., by applying a translation with negative Z value).
- Note that the effect thus produced is **NOT** the same as changing the eye position.



Cont'd

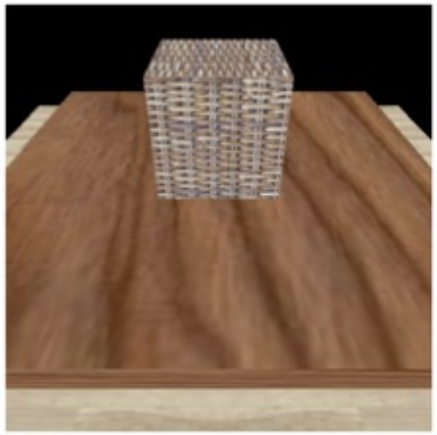
- E.g., (change scene position by translation vector [0.0, 0.0, -5.0])

```
mat4.lookAt([0, 4, 5],[0, 0, 0], [0, 1,0], modelViewMatrix);
mat4.translate(modelViewMatrix, [0.0, 0.0, -5.0], modelViewMatrix);
```

  is **roughly** the same as (change viewer position)

```
mat4.lookAt([0, 4, 10], [0, 0, 0], [0, 1,0], modelViewMatrix);
```

- The reason for the difference in visual effect of the two approaches is that while the scene is moved backward the eye is still looking at (0.0, 0.0, 0.0). To achieve the same result, you need to set `centre` vector at `[0, -5, 0]`



eye= [0,4,5]
Translation= [0.0,0.0,0.0]

eye= [0,4,5]
Translation= [0.0,0.0,-5.0]

eye= [0,4,5]
Translation= [0.0,0.0,-15.0]

Camera Rotation

- A scene can be rotated by applying rotatation transformations. E.g., rotating the scene 15 degrees about x axis (pitch) and then 45 degrees about y (yaw):

```
mat4.lookAt([0, 4, 10], [0, 0, 0], [0, 1,0], modelViewMatrix);
mat4.rotateX(modelViewMatrix, 15.0*Math.PI/180, modelViewMatrix);
mat4.rotateY(modelViewMatrix, 45.0*Math.PI/180, modelViewMatrix);
```

Original scene

Rotation of 15 degs about x

Followed by rotation of 45 degs about y

Interactive Scene Navigation

- To achieve interactive scene navigation at run time, the modelview transformation must be updated with correct values of translations and rotations from user input.
- On standard PCs, this is normally done through handling **mouse** or **key events** by the **event handlers.**
- For this, some global variables can be used to store the values of translation and/or rotations, e.g.,

```
//global variables
Var xRot, yRot, trnasZ; // pitch, yaw, zoom
. . .
function draw(){
     mat4.lookAt([0, 4, 10], [0, 0, 0], [0, 1,0], modelViewMatrix);
     mat4.translate(modelViewMatrix, [0.0, 0.0, transZ], modelViewMatrix);//zoom
     mat4.rotateX(modelViewMatrix, xRot, modelViewMatrix);//pitch
     mat4.rotateY(modelViewMatrix, yRot, modelViewMatrix);// yaw
     . . .
     //draw objects
     . . .
}
```

An Example

- In the table-cube program, we may need to declare these globals:

```
var transZ=0;
var xRot =0.0, yRot =0.0;
```

  where `transZ` controls the amount of zoom. `xRot` and `yRot` control the amounts of rotation about x (pitch) and y axes (yaw).

- We also need to declare the globals for mouse event handling

```
var xOffs = 0.0, yOffs =  0.0;
var drag = 0;
```

  where `xOffs` and `yOffs` store the initial position of mouse when mouse is dragged. The variable `drag` is a flag for implement the mouse drag handler using `mouseMove` event.

Update ModelView Matrix

- In the `draw()` method:

```
mat4.translate(pwgl.modelViewMatrix, [0.0, 0.0,
    transZ], pwgl.modelViewMatrix);
mat4.rotateX(pwgl.modelViewMatrix, xRot,
    pwgl.modelViewMatrix);
mat4.rotateY(pwgl.modelViewMatrix, yRot,
    pwgl.modelViewMatrix);
yRot = xRot =transZ=0;
```

- The `draw()` method is called each frame in the animation loop; therefore, the transformation is updated.
- In this implementation, the *relative* (the existing) translation and rotations are used, therefore `yRot, xRot,transZ` are set zero after translation and rotations being applied.

Mouse Event

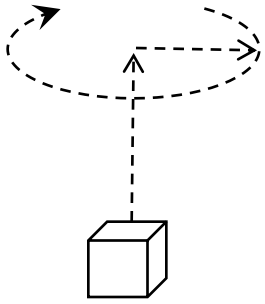
```
function mymousedown( ev ){
  drag  = 1;
  xOffs = ev.clientX;
  yOffs = ev.clientY;
}
```

```
function mymouseup( ev ){
  drag  = 0;
}
```

```
function mymousemove( ev ){
  if ( drag == 0 ) return;
  if ( ev.shiftKey ) {
     transZ = (ev.clientY - yOffs);
  } else  {
    yRot = (ev.clientX - xOffs)*somefactor*Math.PI/180;
    xRot = (ev.clientY - yOffs)*somefactor*Math.PI/180;
  }
    xOffs = ev.clientX;
    yOffs = ev.clientY;
}
```

Animating Objects

- In scene navigation (i.e., camera view control), we have modified the modelview transformation for ***all*** objects (the entire scene).
- If we want an object to move (i.e., change its position), its modelview matrix must be updated at each frame, too.
- Consider, for example, we want the box first move up to a pre-determined height along a straight line then make a circular motion of a fixed or varying radius.



Paths of Objects

- Path control can take place in two forms:
  - objects move along a pre-determined path and no user control is needed; or
  - Paths of objects are interactively controlled by the user.
- In the first case, the position of an object is calculated each frame according to some formula. The amount of motion depends on the time spent on rendering a frame.
- In the case of interactive control, some event handling methods that update translation and rotations must be in place, the same way as we do in the case of control of camera view.

Path Calculation

- To calculate/specify the position of an object at each timestamp along a path, we need to have:
  - the shape/geometry of the path, e.g., a formula/function,
  - the length of the path,
  - the time taken to travel through the path, etc.
- For simple paths, e.g., a constant speed motion along a straight line or a circle, basic skills of geometry and math are enough.
- For for objects moving along generic paths with varying speed, some numeric techniques are needed.

Path Calculation: An Example

- Suppose that we want the cube to move vertically from the tabletop (where y=2.7) to where y=5 in 3 seconds.
- Therefore, the length of the path is (5 - 2.7)=2.3 units
- **Question**: what is the length of a straight line with end points at (x_1,y_1,z_1) and (x_2,y_2,z_2)?
- Although we don't know exactly when the animation will start, we know that at the beginning of the first frame y=2.7.
- What value should we assign to y at the beginning of the 2^{nd}, 3^{rd}, … frame?
- Suppose the cube moves at constant speed and rendering of the first frame takes time dt, at the beginning of the second frame, y should be

$$dt*\{(5\text{-}2.7) \text{ units}/3 \text{ seconds}\}$$

- How can we know dt?

Frame Time

- The time taken to render a frame, dt, varies from frame to frame.
- It depends on several things:
  - the complexity of the scenes,
  - the change of the scene during animation,
  - the viewing angles of a given scene,
  - the hardware and performance of a computer,
  - other applications that run on the same computer at the same time, etc.
- All these mean that we cannot pre-determine an accurate fixed frame time dt.

Cont'd

- In practice, system time is read at the beginning of each frame.
- At the beginning of the first frame, the time is used as the animation start time:

```
currentTime = Date.now();
startTime = currentTime;
```

- At the beginning of the 2nd frame, the current time is read again, so rendering the first frame takes:

```
dt=currentTime-startTime;
startTime = currentTime;
```

- At the beginning of the 3rd frame, the current time is read again, …
- With the elapsed time, dt, known at the beginning of each frame, the position of the object at that moment can be calculated as:

```
y = 2.7 + (5.0-2.7)/3000*(currentTime - startTime);
```

- Note that system time is in milliseconds.

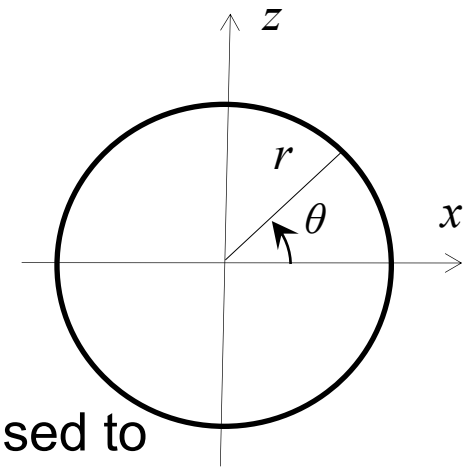
Circular Path

- Similarly, the positions along the circular path can be easily calculated.
- For this, we assume that the time to travel through the circle is 2s.
- Therefore, the angle travelled in dt=(currentTime-startTime) is

  $\theta=(2\pi/2000)*(currentTime-startTime)$

- and the position of the cube is at

  $x=r*cos\theta$, $y=5.0$ and $z=r*sin\theta$



- The code would look like (modulus operator % is used to find the remainder when the angle exceeds 2*PI)

```
angle =2*Math.PI/2000*(currentTime - startTime)%
   (2*Math.PI);
x = Math.cos(angle) * circleRadius;
z = Math.sin(angle) * circleRadius;
```

Update Modelview Matrix

- The calculated x, y and z are used to update the cube's modelview matrix in the method `draw()`:

```
. . .
pushModelViewMatrix();
// calculate a position [x, y, z] on the path

mat4.translate(modelViewMatrix, [x, y, z], modelViewMatrix);
mat4.scale(modelViewMatrix, [0.5, 0.5, 0.5],
    modelViewMatrix);
uploadModelViewMatrixToShader();
drawCube(boxTexture);
popModelViewMatrix();
```

Change Animation Parameters

- Animation parameters, e.g., the radius of the circular path, can be changed in real time from user input.
- Again, it can be done via mouse or key event handling.
- One needs to declare the involved parameter as a global variable:

```
var circleRadius;
```

- In the event handler, this variable is updated on the occurrence of the relevant event, e.g., we may use the arrow up key to increase the radius and the arrow down key to decrease the radius.

```
function handleKeyDown(event) {
   pwgl.listOfPressedKeys[event.keyCode] = true;
}

function handleKeyUp(event) {
   pwgl.listOfPressedKeys[event.keyCode] = false;
}

function handleKeyPress(event) {
}

function handlePressedDownKeys() {
   if (pwgl.listOfPressedKeys[38]) {
         // Arrow up key, increase radius of circle
         circleRadius += 0.1;
   }
   if (pwgl.listOfPressedKeys[40]) {
         // Arrow down key, decrease radius of circle
         circleRadius -= 0.1;
         if (circleRadius < 0) {
                  circleRadius = 0;
         }
   }
}
```

Generic Path

- For a generic path, e.g., a path given by a few sample points in space, the calculation of object positions along the timeline is quite involved.
  - curve fitting – fit your data to a linear or nonlinear curve, e.g., a spline, and calculate the time-position pairs for your animation, or
  - linear or nonlinear interpolation of the control points using splines.
- Such methods are beyond the scope of this unit. If interested, you are welcomed to discuss the topic with me.

Further Reading

- Anyuru, A., WebGL Programming – Develop 3D Graphics for the Web
  - Chapter 6