

02561 Computer Graphics

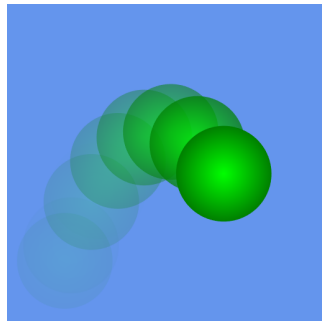
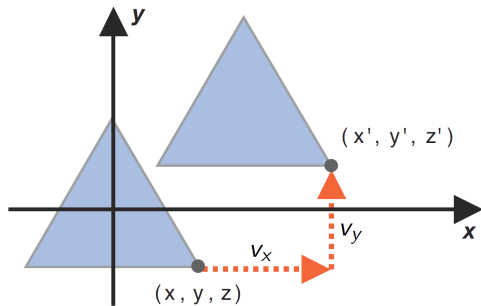
Animation and Interaction

Jeppe Revall Frisvad

September 2020

Making things move

- ▶ We can move things by translation:



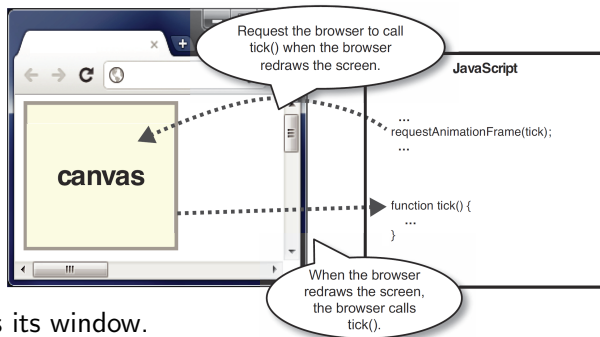
- ▶ Add a translation vector $\mathbf{v} = (v_x, v_y, v_z)$ to all vertex positions $\mathbf{p} = (x, y, z)$ of an object. We can do this in a vertex shader.
- ▶ The `gl_Position` $\mathbf{p}' = (x', y', z')$ returned by the vertex shader is then
$$\mathbf{p}' = \mathbf{p} + \mathbf{v}.$$
- ▶ We make \mathbf{v} available in the vertex shader using a uniform variable.

Drawing repeatedly

- ▶ Draw when resources are available.
- ▶ Avoid queuing up of draw calls. Use:

`requestAnimationFrame(callback);`

where `callback` is a function called when the browser redraws its window.



- ▶ To draw repeatedly, use

```
window.onload = function init()  
{  
  :  
  
  function tick() { render(gl, numPoints); requestAnimationFrame(tick); }  
  tick();  
}  
function render(gl, numPoints)  
{  
  gl.clear(gl.COLOR_BUFFER_BIT);  
  gl.drawArrays(..., 0, numPoints);  
}
```

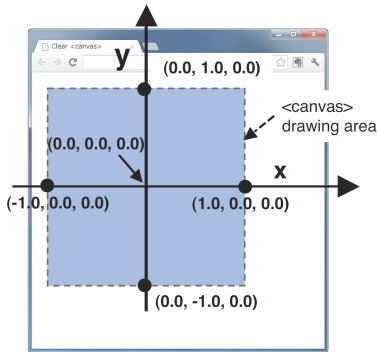
Controlling the movement

- ▶ Use simplistic Newtonian physics.
- ▶ Choose a velocity \mathbf{w} for your moving object.
- ▶ If \mathbf{v} is the translation vector of the object, then for every frame: $\mathbf{v}_{t+1} = \mathbf{v}_t + \mathbf{w}$.
- ▶ The browser refresh rate (frame rate) is commonly 60 fps (frames per second).
- ▶ Recall the default coordinate system.
- ▶ If $w_y = 1$ units per frame, the object will be moving through 30 canvases per second!
- ▶ Moving through an 0.3 part of the canvas ($w_y = 0.01$) is usually more appropriate.
- ▶ Moving back and forth:

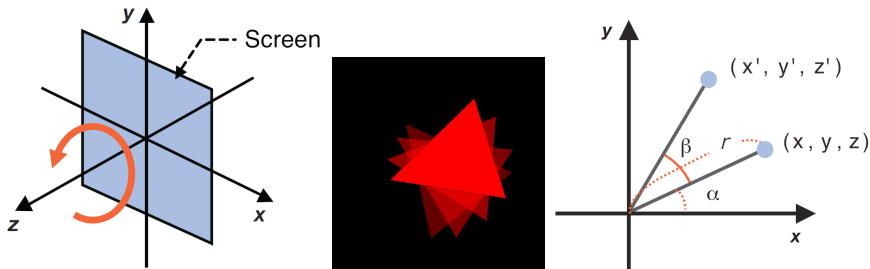
$$\mathbf{v}_{t+1} = \mathbf{v}_t + \mathbf{w}_t$$

$$\mathbf{w}_{t+1} = \text{sgn}(1 - r - |\mathbf{v}_t|) \mathbf{w}_t.$$

where r is for example the radius of the object.



Rotational movement (2D)



- ▶ In 2D, we usually rotate around the z-axis.
- ▶ We can then work in polar coordinates: $\mathbf{p} = (x, y, z) = (r \cos \alpha, r \sin \alpha, z)$.
- ▶ Then, after rotation by the angle β , we have (using angle addition formulae)

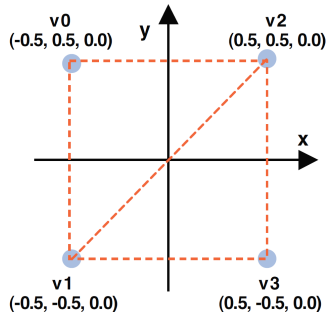
$$\mathbf{p}' = \begin{bmatrix} r \cos(\alpha + \beta) \\ r \sin(\alpha + \beta) \\ z \end{bmatrix} = r \begin{bmatrix} \cos \alpha \cos \beta - \sin \alpha \sin \beta \\ \sin \alpha \cos \beta + \cos \alpha \sin \beta \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ z \end{bmatrix} = \cos \beta \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} + \sin \beta \begin{bmatrix} -y \\ x \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ z \end{bmatrix}.$$

- ▶ We make β available in the vertex shader using a uniform variable.

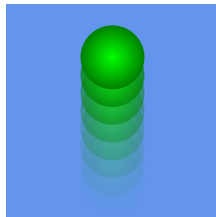
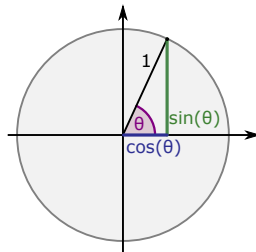
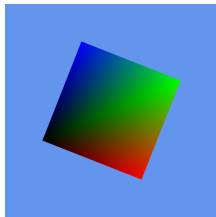
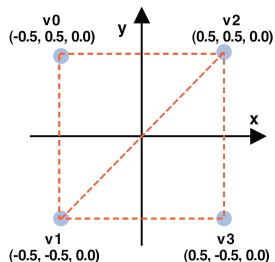
Orbital angular velocity

- ▶ Choose an orbital angular velocity ω for rotational movement: $\beta_{t+1} = \beta_t + \omega$.
- ▶ Consider a square with $x \in [-\frac{1}{2}, \frac{1}{2}]$ and $y \in [-\frac{1}{2}, \frac{1}{2}]$.
- ▶ The radius of its circumcircle is $r = \frac{\sqrt{2}}{2}$.
- ▶ The browser frame rate is commonly 60 fps.
- ▶ For $\omega = 1$ radians per frame, the square would do $\frac{60}{2\pi} \approx 9.5$ full rotations per second.
- ▶ The vertices would move $60 \frac{\sqrt{2}}{2} \approx 42$ distance units (21 canvases) per second.
- ▶ Around one tenth of a full rotation per second ($\omega = 0.01$) is usually more appropriate.
- ▶ We could then implement our tick function as follows:

```
var betaLoc = gl.getUniformLocation(program, "beta");
var beta = 0.0;
function tick() {
    beta += 0.01;          gl.uniform1f(betaLoc, beta);
    render(gl, numPoints); requestAnimationFrame(tick);
}
```



Exercises: simple animation (W01P4, W01P5)



- ▶ Draw repeatedly (`requestAnimationFrame`).
- ▶ Draw a quad (`gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, or `gl.TRIANGLE_FAN`).
- ▶ Use a uniform angle (β) and implement rotational motion in the vertex shader.
- ▶ Draw a circle (`gl.TRIANGLE_FAN`, see slide from Week 1).
- ▶ Use a uniform translation vector (\mathbf{v} , or offset v_y) and implement back and forth motion in the vertex shader.

Interaction through input events



- ▶ Event-handling callback functions: “event listeners”.
- ▶ Event listeners can be added to any element of a webpage.
- ▶ Example (listen for canvas mouse clicks):

- ▶ In webpage HTML source

```
<canvas id="webgl" width="512" height="512">Please use a browser that supports HTML5 canvas.</canvas>
```

- ▶ In JavaScript init function

```
var canvas = document.getElementById("webgl");
canvas.addEventListener("click", function (ev) {

    :    // my callback function

});
```



- ▶ The list of events that we can listen for is extensive. Examples:

click, mousedown, mouseup, mousemove, keypress, keydown, keyup, touchstart, touchmove, touchend, ...

- https://www.w3schools.com/jsref/dom_obj_event.asp

Position input

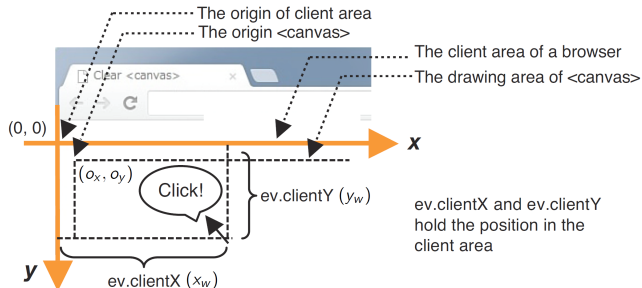
- ▶ Let's make the bouncing ball chase the mouse.
- ▶ We need the position of each mouse event. [A: 3.7]

$$\left(2 \frac{x_w}{w} - 1, 2 \frac{h - y_w - 1}{h} - 1\right)$$

```
var offset = vec2(0.0, 0.0); // v_t
var velocity = vec2(0.0, 0.0); // w_t
var mousepos = vec2(0.0, 0.0);
canvas.addEventListener("mousemove", function (ev) {
  mousepos = vec2(2*ev.clientX/canvas.width - 1, 2*(canvas.height - ev.clientY - 1)/canvas.height - 1);
  velocity = vec2((mousepos[0] - offset[0])*speed, (mousepos[1] - offset[1])*speed);
});
```

- ▶ Shift of origin between client area and canvas: $\left(2 \frac{x_w - o_x}{w} - 1, 2 \frac{h - (y_w - o_y) - 1}{h} - 1\right)$
- ▶ Retrieve the bounding box of the canvas in the client area and use it for correction:

```
canvas.addEventListener("mousemove", function (ev) {
  var bbox = ev.target.getBoundingClientRect();
  mousepos = vec2(2*(ev.clientX - bbox.left)/canvas.width - 1, 2*(canvas.height - ev.clientY + bbox.top - 1)/canvas.height - 1);
  velocity = vec2((mousepos[0] - offset[0])*speed, (mousepos[1] - offset[1])*speed);
});
```



Vertex buffer pre-allocation

- ▶ Suppose we would like to draw a point for every mouse click.
- ▶ GPU memory allocation for buffer data is expensive. So we should pre-allocate.
- ▶ When the data is not known in advance, we choose a buffer size (in bytes).
- ▶ In the init function

```
var max_verts = 1000;  
var index = 0; var numPoints = 0;  
var vBuffer = gl.createBuffer();  
gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);  
gl.bufferData(gl.ARRAY_BUFFER, max_verts*sizeof['vec2'], gl.STATIC_DRAW);
```

- ▶ We can then later insert data from a position vector p at an index.
- ▶ In an event callback function

```
gl.bufferSubData(gl.ARRAY_BUFFER, index*sizeof['vec2'], flatten(p));  
numPoints = Math.max(numPoints, ++index); index %= max_verts;
```

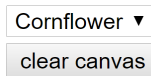
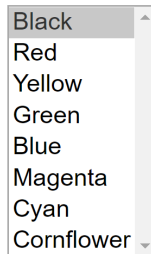
Buttons and selection menus

- ▶ Create a user interface for your program using HTML.
- ▶ Add buttons and selection menus.

```
<select id="colorMenu" size="8">  
  <option value="0" selected>Black</option>  
  .  
  .  
  <option value="7">Cornflower</option>  
</select>  
<select id="clearMenu" size="1">  
  <option value="0">Black</option>  
  .  
  .  
  <option value="7" selected>Cornflower</option>  
</select><br/>  
<button id="clearButton">clear canvas</button>
```

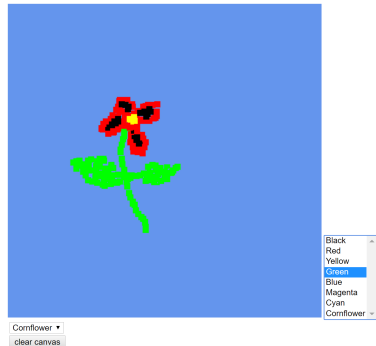
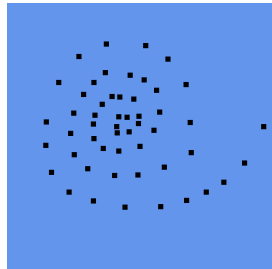
- ▶ Define a colors array in JavaScript [A: 3.10], then

```
var clearMenu = document.getElementById("clearMenu");  
var clearButton = document.getElementById("clearButton");  
clearButton.addEventListener("click", function(event) {  
  var bgcolor = colors[clearMenu.selectedIndex];  
  gl.clearColor(bgcolor[0], bgcolor[1], bgcolor[2], bgcolor[3]);  
  .  
  .  
});
```



Exercises: simple interaction (W02P1, W02P2)

- ▶ Pre-allocate a buffer for vertex positions.
- ▶ Add an event listener to the canvas retrieving the position of mouse clicks.
- ▶ Insert coordinates of each clicked point into the pre-allocated buffer.
- ▶ Pre-allocate a buffer for vertex colors.
- ▶ Add selection menus and clear button and associated click event listener(s).
- ▶ Insert the currently selected RGBA color into the pre-allocated color buffer for each clicked point.
- ▶ Let the clear button clear the canvas with the currently selected clear color (restart insertion of points at index 0).



Index arrays, buffer offsets, and draw modes

- ▶ With one large vertex buffer, we can draw many different shapes.

```
gl.drawArrays(draw_mode, index_offset, no_of_vertices);
```

- ▶ We need index arrays to tell us where the vertices of the different shapes are placed in the buffer.

```
var points = []; var triangles = []; var circles = [];
```

- ▶ Use `points.push(index)` to store indices of the vertices to be drawn as points.
- ▶ When drawing triangles, we can for every third point do

```
points.pop(); triangles.push(points.pop());
```

- ▶ When drawing circles, we can for every second point do

```
circles.push(points.pop());
```

