

**PROJECT 1**

PROFESSOR: Prof. Hiqmet Kamberaj

STUDENT: Adelina Klica

EXAM: Midterm Exam

21.03.2021

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Hierarchical Modeling 2D</title>

<link rel="stylesheet" href="../demo.css">

<script src="../script/demo-core.js"></script>

<script>var canvas;

var graphics;

// ---------------- The object-oriented scene graph API ------------------

/\*\*

\* The base class for all nodes in the scene graph data structure.

\*/

function SceneGraphNode() {

this.fillColor = null; // If non-null, the default fillStyle for this node.

this.strokeColor = null; // If non-null, the default strokeStyle for this node.

}

SceneGraphNode.prototype.doDraw = function(g) {

// This method is meant to be abstract and must be

// OVERRIDDEN in any actual object in the scene graph.

// It is not meant to be called; it is called by draw().

throw "doDraw not implemented in SceneGraphNode"

}

SceneGraphNode.prototype.draw = function(g) {

// This method should be CALLED to draw the object

// represented by this SceneGraphNode. It should NOT

// ordinarily be overridden in subclasses.

graphics.save();

if (this.fillColor) {

g.fillStyle = this.fillColor;

}

if (this.strokeColor) {

g.strokeStyle = this.strokeColor;

}

this.doDraw(g);

graphics.restore();

}

SceneGraphNode.prototype.setFillColor = function(color) {

// Sets fillColor for this node to color.

// Color should be a legal CSS color string, or null.

this.fillColor = color;

return this;

}

SceneGraphNode.prototype.setStrokeColor = function(color) {

// Sets strokeColor for this node to color.

// Color should be a legal CSS color string, or null.

this.strokeColor = color;

return this;

}

SceneGraphNode.prototype.setColor = function(color) {

// Sets both the fillColor and strokeColor to color.

// Color should be a legal CSS color string, or null.

this.fillColor = color;

this.strokeColor = color;

return this;

}

/\*\*

\* Defines a subclass, CompoundObject, of SceneGraphNode to represent

\* an object that is made up of sub-objects. Initially, there are no

\* sub-objects.

\*/

function CompoundObject() {

SceneGraphNode.call(this); // do superclass initialization

this.subobjects = []; // the list of sub-objects of this object

}

CompoundObject.prototype = new SceneGraphNode(); // (makes it a subclass!)

CompoundObject.prototype.add = function(node) {

// Add node a subobject of this object. Note that the

// return value is a reference to this node, to allow chaining

// of method calls.

this.subobjects.push(node);

return this;

}

CompoundObject.prototype.doDraw = function(g) {

// Just call the sub-objects' draw() methods.

for (var i = 0; i < this.subobjects.length; i++)

this.subobjects[i].draw(g);

}

/\*\*

\* Define a subclass, TransformedObject, of SceneGraphNode that

\* represents an object along with a modeling transformation to

\* be applied to that object. The object must be specified in

\* the constructor. The transformation is specified by calling

\* the setScale(), setRotate() and setTranslate() methods. Note that

\* each of these methods returns a reference to the TransformedObject

\* as its return value, to allow for chaining of method calls.

\* The modeling transformations are always applied to the object

\* in the order scale, then rotate, then translate.

\*/

function TransformedObject(object) {

SceneGraphNode.call(this); // do superclass initialization

this.object = object;

this.rotationInDegrees = 0;

this.scaleX = 1;

this.scaleY = 1;

this.translateX = 0;

this.translateY = 0;

}

TransformedObject.prototype = new SceneGraphNode(); // (makes it a subclass!)

TransformedObject.prototype.setRotation = function(angle) {

// Set the angle of rotation, measured in DEGREES. The rotation

// is always about the origin.

this.rotationInDegrees = angle;

return this;

}

TransformedObject.prototype.setScale = function(sx, sy) {

// Sets scaling factors.

this.scaleX = sx;

this.scaleY = sy;

return this;

}

TransformedObject.prototype.setTranslation = function(dx,dy) {

// Set translation mounts.

this.translateX = dx;

this.translateY = dy;

return this;

}

TransformedObject.prototype.doDraw = function(g) {

// Draws the object, with its modeling transformation.

g.save();

if (this.translateX != 0 || this.translateY != 0) {

g.translate(this.translateX, this.translateY);

}

if (this.rotationInDegrees != 0) {

g.rotate(this.rotationInDegrees/180\*Math.PI);

}

if (this.scaleX != 1 || this.scaleY != 1) {

g.scale(this.scaleX, this.scaleY);

}

this.object.draw(g);

g.restore();

}

/\*\*

\* A subclass of SceneGraphNode representing filled triangles.

\* The constructor specifies the vertices of the triangle:

\* (x1,y1), (x2,y2), and (x3,y3).

\*/

function Triangle(x1,y1,x2,y2,x3,y3) {

SceneGraphNode.call(this);

this.x1 = x1;

this.y1 = y1;

this.x2 = x2;

this.y2 = y2;

this.x3 = x3;

this.y3 = y3;

}

Triangle.prototype = new SceneGraphNode();

Triangle.prototype.doDraw = function(g) {

g.beginPath();

g.moveTo(this.x1,this.y1);

g.lineTo(this.x2,this.y2);

g.lineTo(this.x3,this.y3);

g.closePath();

g.fill();

}

/\*\*

\* Directly create a line object as a SceneGraphNode with a

\* custom doDraw() method. line is of length 1 and

\* extends along the x-axis from (0,0) to (1,0).

\*/

var line = new SceneGraphNode();

line.doDraw = function(g) {

g.beginPath();

g.moveTo(0,0);

g.lineTo(1,0);

g.stroke();

}

/\*\*

\* Directly create a filled rectangle object as a SceneGraphNode with a

\* custom doDraw() method. filledRect is a square with side 1, centered

\* at (0,0), with corners at (-0.5,-0.5) and (0.5,0.5).

\*/

var filledRect = new SceneGraphNode();

filledRect.doDraw = function(g) {

g.fillRect(-0.5,-0.5,1,1);

}

/\*\*

\* Directly create a rectangle object as a SceneGraphNode with a

\* custom doDraw() method. rect is a square with side 1, centered

\* at (0,0), with corners at (-0.5,-0.5) and (0.5,0.5). Only the

\* outline of the square is drawn.

\*/

var rect = new SceneGraphNode();

rect.doDraw = function(g) {

g.strokeRect(-0.5,-0.5,0,1);

}

/\*\*

\* Directly create a filled circle object as a SceneGraphNode with a

\* custom doDraw() method. filledCircle is a circle with radius 0.5

\* (diameter 1), centered at (0,0).

\*/

var filledCircle = new SceneGraphNode();

filledCircle.doDraw = function(g) {

g.beginPath();

g.arc(0,0,0.5,0,2\*Math.PI);

g.fill();

}

/\*\*

\* Directly create a circle object as a SceneGraphNode with a

\* custom doDraw() method. filledCircle is a circle with radius 0.5

\* (diameter 1), centered at (0,0). Only the outline of the circle

\* is drawn.

\*/

var circle = new SceneGraphNode();

circle.doDraw = function(g) {

g.beginPath();

g.arc(0,0,0.5,0,2\*Math.PI);

g.stroke();

}

// -------------------- Specific to this application ----------------------------

/\*

\* Define two extra basic objects as SceneGraphNodes with custom doDraw() methods.

\* One represents the ground, the other a vane for a windmill.

\*/

var ground = new SceneGraphNode();

ground.doDraw = function(g) {

g.beginPath();

g.moveTo(0, -1);

g.lineTo(0, 0.8);

g.lineTo(0, 1.5);

g.lineTo(0.1, 1.5);

g.lineTo(0.1, 1.8);

g.lineTo(0.2, 1.8);

g.lineTo(0.4, 2.5);

g.lineTo(0.6, 1.8);

g.lineTo(0.8, 1.8);

g.lineTo(0.8, 1.5);

g.lineTo(0.9, 1.5);

g.lineTo(0.9, 1);

g.lineTo(1.5, 1);

g.lineTo(1.2, 1);

g.lineTo(1.2, 2);

g.lineTo(1.3, 2);

g.lineTo(1.4, 3);

g.lineTo(1.5, 2);

g.lineTo(1.6, 2);

g.lineTo(1.6, 1.5);

g.lineTo(1.8, 1.5);

g.lineTo(1.8, 1.2);

g.lineTo(2.2, 1.2);

g.lineTo(2.2, 0.8);

g.lineTo(2.2, 1.8);

g.lineTo(2.5, 1.8);

g.lineTo(2.5, 1);

g.lineTo(2.7, 1);

g.lineTo(2.7, 1.5);

g.lineTo(3, 1.5);

g.lineTo(3, 2);

g.lineTo(3.2, 2.3);

g.lineTo(3.4, 2);

g.lineTo(3.4, 1);

g.lineTo(3.5, 1);

g.lineTo(3.5, 1.7);

g.lineTo(3.7, 2);

g.lineTo(3.9, 1.7);

g.lineTo(3.9, 1.4);

g.lineTo(4.4, 1.4);

g.lineTo(4.4, 1.2);

g.lineTo(4.6, 1.2);

g.lineTo(4.6, 1);

g.lineTo(4.8, 1);

g.lineTo(4.8, 1.7);

g.lineTo(5, 1.7);

g.lineTo(5, 2);

g.lineTo(5.3, 2);

g.lineTo(5.3, 1.4);

g.lineTo(5.5, 1.4);

g.lineTo(5.5, 1.7);

g.lineTo(6, 1.7);

g.lineTo(6, 1);

g.lineTo(6.2, 1);

g.lineTo(6.2, 1.5);

g.lineTo(6.3, 1.5);

g.lineTo(6.4, 2);

g.lineTo(6.5, 2);

g.lineTo(6.9, 2);

g.lineTo(7, 2);

g.lineTo(7, -1);

g.closePath();

g.fill();

}

var windmillVane = new SceneGraphNode();

windmillVane.doDraw = function(g) {

g.beginPath();

g.moveTo(0,0);

g.lineTo(0.1, 0.1);

g.lineTo(0.5,0);

g.lineTo(0.1, -0.1);

g.closePath();

g.fill();

}

var world; // A SceneGraphNode representing the entire picture. This should

// be created in the createWorld() method.

var pixelSize; // The size of one pixel, in the transformed coordinates.

// This is used as the default width of a stroke.

var background = "#00266D"// A CSS color string giving the background color.

// the draw() function fills the canvas with this color.

var xleft = 0; // The requested xy-limits on the canvas, after the

var xright = 7; // coordinate transformation has been applied.

var ybottom = -1; // The transformation is applied in the draw() function.

var ytop = 4;

var frameNumber = 0; // Current frame number.

var cart; // TransformedObjects that are animated.

var cart2;

var bicycle;

var wheel;

var sun;

var rotor;

/\*\*

\* Create the scene graph data structure. The global variable world must

\* refer to the root node of the scene graph. This function is called in

\* the init() function.

\*/

function createWorld() {

var i;

var sunTemp = new CompoundObject();

sunTemp.setColor("#DFDFDF"); // color for filled circle

for (i = 0; i < 12; i++) { // add the 12 light rays, with different rotations, colored orange

sunTemp.add(new TransformedObject(line).setScale(0.0, 0.75).setRotation(i \* 30).setColor("#8a857f") );

}

sunTemp.add( filledCircle ); // the face of the sun

sunTemp.add(new TransformedObject(circle).setColor("#FFFFFF") ); // outlines the face //BOJA OBRUCA

sun = new TransformedObject(sunTemp);

var wheelTemp = new CompoundObject();

wheelTemp.setColor("black"); // color for all but one of the subobjects

wheelTemp.add( new TransformedObject(filledCircle).setScale(2,2) );

wheelTemp.add(new TransformedObject(filledCircle).setScale(1.6, 1.6).setColor("#7F7F7F") ); //BOJA GUMA

wheelTemp.add( new TransformedObject(filledCircle).setScale(0.4,0.4) );

for (i = 0; i < 12; i++) { // add the 12 spokes

wheelTemp.add( new TransformedObject(line).setRotation(i\*30) );

}

wheel = new TransformedObject(wheelTemp);

var cartTemp = new CompoundObject();

cartTemp.setColor("#40FF8E"); // color for the rectangular body of the cart //CART 1

cartTemp.add(new TransformedObject(wheel).setScale(0.8, 0.8).setTranslation(1.65, -0.1));

cartTemp.add(new TransformedObject(wheel).setScale(0.8, 0.8).setTranslation(-1.65, -0.1));

cartTemp.add(new TransformedObject(wheel).setScale(0.8, 0.8).setTranslation(2, -0.1));

cartTemp.add(new TransformedObject(wheel).setScale(0.8, 0.8).setTranslation(-1.4, -0.1));

cartTemp.add(new TransformedObject(filledRect).setScale(6, 1.5).setTranslation(0, 1)); // the body of the cart

cartTemp.add(new TransformedObject(filledRect).setScale(2.6, 1).setTranslation(-1, 2)); // the top of the cart

cart = new TransformedObject(cartTemp).setScale(0.3, 0.3);

var cartTemp2 = new CompoundObject();

cartTemp2.setColor("#1DCAC2"); // color for the rectangular body of the cart //CART 2

cartTemp2.add(new TransformedObject(wheel).setScale(0.8, 0.8).setTranslation(1.65, -0.1));

cartTemp2.add(new TransformedObject(wheel).setScale(0.8, 0.8).setTranslation(-1.65, -0.1));

cartTemp2.add(new TransformedObject(wheel).setScale(0.8, 0.8).setTranslation(2, -0.1));

cartTemp2.add(new TransformedObject(wheel).setScale(0.8, 0.8).setTranslation(-1.4, -0.1));

cartTemp2.add(new TransformedObject(filledRect).setScale(6, 1.5).setTranslation(0, 1)); // the body of the cart

cartTemp2.add(new TransformedObject(filledRect).setScale(3, 2).setTranslation(-1, 2)); // the top of the cart

cart2 = new TransformedObject(cartTemp2).setScale(0.3, 0.3);

var cartTempb = new CompoundObject();

cartTempb.setColor("#FF0000"); // color for the rectangular body of the cart //BICYCLE

cartTempb.add(new TransformedObject(wheel).setScale(0.8, 0.8).setTranslation(1.65, -0.1));

cartTempb.add(new TransformedObject(wheel).setScale(0.8, 0.8).setTranslation(-1.65, -0.1));

cartTempb.add(new TransformedObject(filledRect).setScale(0.3, 1.8).setTranslation(-1.6 ,0.9)); // back wheel body

cartTempb.add(new TransformedObject(filledRect).setScale(1.5, 0.5).setTranslation(-1.5, 2)); //the top of the bicycle

cartTempb.add(new TransformedObject(filledRect).setScale(0.3, 3).setTranslation(1.6, 1.5)); //front wheel body

cartTempb.add(new TransformedObject(filledRect).setScale(3.4, 0.2).setTranslation(-0.1, 0.9)); //connects two wheels

cartTempb.add(new TransformedObject(filledRect).setScale(1.4, 0.2).setTranslation(1.6, 3)); //steering wheel

cartTempb.add(new TransformedObject(filledRect).setScale(1, 0.5).setTranslation(-0.1, 0.9));

bicycle = new TransformedObject(cartTempb).setScale(0.25, 0.25);

var rotorTemp = new CompoundObject(); // a "rotor" consisting of three vanes

rotorTemp.setColor( "#FAFF90" ); // color for all of the vanes

rotorTemp.add( windmillVane );

rotorTemp.add( new TransformedObject(windmillVane).setRotation(70) ); //UDALJENOST OR SMTH

rotorTemp.add(new TransformedObject(windmillVane).setRotation(140));

rotorTemp.add(new TransformedObject(windmillVane).setRotation(210));

rotorTemp.add(new TransformedObject(windmillVane).setRotation(280));

rotor = new TransformedObject(rotorTemp);

var windmill = new CompoundObject();

windmill.setColor("#AA9999"); // color for the pole

//windmill.add( new TransformedObject(filledRect).setScale(0.1,3).setTranslation(0,1.5) ); // the pole

windmill.add( new TransformedObject(rotor).setTranslation(0,4) ); // the rotating vanes

world = new CompoundObject();

world.setColor("#000000"); // color used for the ground only

world.add(ground);

world.add( new TransformedObject(filledRect).setScale(7,0.8).setTranslation(3.5,0).setColor("#646496") );

world.add( new TransformedObject(filledRect).setScale(7,0.06).setTranslation(3.5,0).setColor("white") );

world.add( new TransformedObject(windmill).setScale(0.3,0.3).setTranslation(0.75,2) );

world.add( new TransformedObject(windmill).setScale(0.3,0.3).setTranslation(2.2,2.3) );

world.add(new TransformedObject(windmill).setScale(0.3, 0.3).setTranslation(3.7, 2.0));

world.add(new TransformedObject(windmill).setScale(0.3, 0.3).setTranslation(2.5, 1.3));

world.add(new TransformedObject(windmill).setScale(0.3, 0.3).setTranslation(4.7, 1.2)); //DODAJ ZVJEZDICE

world.add(new TransformedObject(windmill).setScale(0.3, 0.3).setTranslation(6.0, 1.3));

world.add( new TransformedObject(sun).setTranslation(5.5,3.3) );

world.add(cart);

world.add(cart2);

world.add(bicycle);

}

/\*\*

\* This will be called before each frame is drawn.

\*/

function updateFrame() {

frameNumber++;

cart.setTranslation(-3 + 13 \* (frameNumber % 300) / 300.0, 0);

cart2.setTranslation(-1 + 13 \* (frameNumber % 300) / 300.0, 0);

bicycle.setTranslation(-5 + 13 \* (frameNumber % 300) / 300.0, 0);

wheel.setRotation(-frameNumber\*3.1);

sun.setRotation(-frameNumber);

rotor.setRotation(frameNumber \* 2.7);

}

// ------------------------------- graphics support functions --------------------------

/\*\*

\* Draw one frame of the animation. Probably doesn't need to be changed,

\* except maybe to change the setting of preserveAspect in applyLimits().

\*/

function draw() {

graphics.save(); // to make sure changes don't carry over from one call to the next

graphics.fillStyle = background; // background color

graphics.fillRect(0,0,canvas.width,canvas.height);

graphics.fillStyle = "black";

applyLimits(graphics,xleft,xright,ytop,ybottom,true);

graphics.lineWidth = pixelSize;

world.draw(graphics);

graphics.restore();

}

/\*\*

\* Applies a coordinate transformation to the graphics context, to map

\* xleft,xright,ytop,ybottom to the edges of the canvas. This is called

\* by draw(). This does not need to be changed.

\*/

function applyLimits(g, xleft, xright, ytop, ybottom, preserveAspect) {

var width = canvas.width; // The width of this drawing area, in pixels.

var height = canvas.height; // The height of this drawing area, in pixels.

if (preserveAspect) {

// Adjust the limits to match the aspect ratio of the drawing area.

var displayAspect = Math.abs(height / width);

var requestedAspect = Math.abs(( ybottom-ytop ) / ( xright-xleft ));

var excess;

if (displayAspect > requestedAspect) {

excess = (ybottom-ytop) \* (displayAspect/requestedAspect - 1);

ybottom += excess/2;

ytop -= excess/2;

}

else if (displayAspect < requestedAspect) {

excess = (xright-xleft) \* (requestedAspect/displayAspect - 1);

xright += excess/2;

xleft -= excess/2;

}

}

var pixelWidth = Math.abs(( xright - xleft ) / width);

var pixelHeight = Math.abs(( ybottom - ytop ) / height);

pixelSize = Math.min(pixelWidth,pixelHeight);

g.scale( width / (xright-xleft), height / (ybottom-ytop) );

g.translate( -xleft, -ytop );

}

//------------------ Animation framework ------------------------------

var animationTimeout = null; // A null value means the animation is off.

// Otherwise, this is the timeout ID.

function frame() {

// Draw one frame of the animation, and schedule the next frame.

updateFrame();

draw();

animationTimeout = setTimeout(frame, 33);

}

function setAnimationRunning(run) {

if ( run ) {

if (animationTimeout == null) {

// If the animation is not already running, start

// it by scheduling a call to frame().

animationTimeout = setTimeout(frame, 33);

}

}

else {

if (animationTimeout != null) {

// If the animation is running, stop it by

// canceling the next scheduled call to frame().

clearTimeout(animationTimeout);

}

animationTimeout = null; // Indicates that animation is off.

}

}

//----------------------- initialization -------------------------------

function init() {

canvas = document.getElementById("maincanvas");

if (!canvas.getContext) {

document.getElementById("headline").innerHTML = "ERROR: Canvas not supported";

return;

}

graphics = canvas.getContext("2d");

document.getElementById("animateCheck").checked = true; // Make sure box is checked!

createWorld();

setAnimationRunning(true); // start the animation

}</script>

</head>

<body onload="init()">

<div id="content">

<h3 id="headline">Hierarchical Modeling 2D</h3>

<div id="canvas-holder">

<canvas id="maincanvas" width="420" height="300"></canvas>

</div>

<br style="clear:both">

<p>

<label>

<input type="checkbox" id="animateCheck"

onchange="setAnimationRunning(this.checked)">Run the Animation

</label>

</p>

</div>

<div id="help-content" style="display:none">

<h3>About this demo...</h3>

<p>

This demo shows a simple animated 2D scene. The scene is constructed

using hierarchical modeling. That is objects, such as the cart,

can be made up of simpler objects, such as wheels, which can in

turn be made up of even simpler objects, such as lines and disks,

and so on. (The source code for this demo uses an object-oriented

scene graph API.)

</p>

<p>

The only interactivity is the checkbox, which turns the

animation off and on.

</p>

</div>

<!-- support for help text -- do not change. -->

<div id="help-icon">

<img src="question32.png" onclick="showDemoHelp()"

title="Click here for information about this demo." width="32" height="32">

</div>

<div id="hide-help-icon">

<img src="close32.png" onclick="showDemoHelp()"

title="Click here to return to the demo." width="65" height="32">

</div>

<div id="helpBG" style="display:none"></div>

</body>

</html>

