Assignment 2 – Geometry and Animation

UPR Río Piedras, Computer Science – Computer Graphics, Spring 2019

Abstract: Implement geometric modeling and animations using the 2D HTML5 canvas.

Guidelines for assignment hand in:
- Upload one ZIP file on Moodle, with extension .zip (please, no RAR or 7zip files)
- This ZIP file contains all code and data necessary to run your assignment. It is organized as follows
- Program should be using the 2D HTML5 canvas with Javascript.
- File organization (filenames, flat or hierarchical subfolders…) is up to you, but should be clear and meaningful.
- Make sure the example run by simply opening the HTML file, without having to modify the code. If any additional action is required to run the results, this must be explained in a clear way directly when opening the HTML page in the browser, with additional details in a README.txt if needed (for instance, if it needs to run through a webserver instead of directly from the file).

Part 0: Questions [15]

Answer these questions in a Questions.pdf or Questions.html document. A detailed answer is expected, with diagrams to support your explanations.

Q1. [5] Give a geometric interpretation of the point (x0+sin θ, y0+cos θ) . Make sure your explanation is complete.

Q2. [5] Explain what are homogeneous coordinates and homogeneous transformation matrix and why they are useful.

Q3. [5] Explain the concept of a scenegraph and why it is useful.

Implementation, Part 1 and 2:

Documentation [10]
- Your code should be well organized (pay attention to indenting) and commented (Include comments that specify any non obvious convention or intent).
- The HTML page should provide below the canvas a short but clear explanation for the user to know how to access all the functionalities you developed.

Do not forget to provide description of the implemented features directly on the HTML page so that the user can discover them.

Part 1: Animation and keyboard [45]

Objective: Display animated objects and control them using the keyboard, similar to Asteroids game.

Use template code `asteroid.html` that contains a canvas and a GUI to help debugging by enabling the visualization and modification of the parameters.
You need to submit only the final code, as the questions build on each other.

Q1) Ship drawing [15]
a) Transform the drawing of the asteroid spaceship in `drawShip()` so that it is centered on (x0, y0) and has the correct angle (see Figure 1). Use only `ctx.translate` and `ctx.rotate` for that.
b) Replace the wireframe of the spaceship by a nice image of spaceship. This image should be centered on the wireframe, take roughly the same amount of space, and rotate and translate in the exact same way as the wireframe did.

Hints:
a) Use the methodology discussed in class to decide in which order translation and rotation should be applied. When drawing, the ship is centered at (0,0) in ship coordinates, but the transforms move that to canvas coordinates. The center of the ship should NOT change when rotating. Be careful that `params.angle` is expressed in degrees, but `ctx.rotate` expects radians. You can test your function by changing the parameters in the parameters GUI.
b) You may need to use negative coordinates to center the ship image properly on position (0,0).
The `drawShip` function should read all information from the Model to respect MVC philosophy.

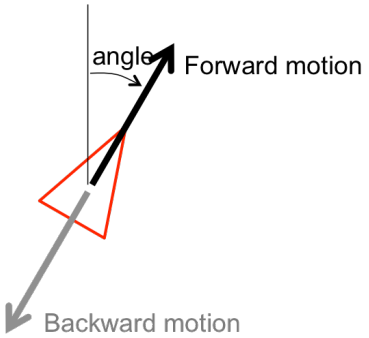

Figure 1: Specification of angle for spaceship orientation and forward/backward motion.

Q2) Ship control [20]
Use keyboard polling to check if an arrow key or motion key is down at each animation frame. If an arrow is down, do the following:
- **up arrow/i**: accelerate the ship using a fixed acceleration in the forward direction
- **down arrow/k**: accelerate the ship using a fixed acceleration in the backward direction
- **right arrow/l**: rotate the ship clockwise using a fixed angular velocity
- **left arrow/j**: rotate the ship counter-clockwise using a fixed angular velocity

When the ship exits the image, it is teleported to the opposite border, following a 2D torus topology (right ↔ left, top↔bottom)

Hints:
To use polling, we need to first convert the event based keyboard I/O into polling based I/O:
Use the `onKeyDown` callback to update a variable that stores the current state of the keyboard, check `event.key` or `event.keyCode` to identify which key was pressed. Define a `isKeyDown` function that checks this state from the `tick` function.
To help debugging, you may first try simple ship movements inside `onKeyDown`, but for the purpose of animation, polling is needed.

If we denote by (dx,dy) the unit vector in the forward direction (defined using Math.cos and Math.sin), then when **up arrow** is pressed, the acceleration is defined as:

```
ax = dx * fixedAcceleration
ay = dy * fixedAcceleration
```

Then, velocity is updated by integrating acceleration over the time interval since last update

```
vx = vx + ax * elapsed
vy = vy + ay * elapsed
```

and position is updated by integrating velocity

```
x0 = x0 + vx * elapsed
y0 = x0 + vy * elapsed
```

where `elapsed` is the amount of time since last update (measured in seconds).
Therefore, the units of the vectors are:

- position (x0,y0) in pixels
- velocity (vx,vy) in pixels/s (displacement during 1s)
- acceleration (ax,ay) in pixels/s^2 (change in velocity if we press the up arrow during 1s)

If no key is pressed, the acceleration should be zero. In that case, the velocity `(vx,vy)` will not change, and the ship will continue to move following a straight line at constant speed.

The teleportation simply requires comparing x0 and y0 to each of the 4 sides of the canvas and adding/subtracting the width or height of the canvas to put the ship back in the visible area.

For the angle, this is simply a fixed velocity

```
angle = angle + angleVelocity * elapsed
```

Therefore, the units of the angular variables are:

- angular position angle in deg
- angular velocity angleVelocity in deg/s (angle change if we press right arrow during 1s)

Note: The fixed velocity for rotation is not physically realistic in terms of spaceship dynamics, but improves the "playability" of the game.

`drawAll` should be called automatically at the end of each `tick`, so you just need to modify the model stored in `params` to update the ship display (following the MVC pattern).

Q3) Bullet [10]

When the user presses key Space (`onKeyDown`), fire a bullet from the spaceship in the forward direction. The bullet should move with fixed velocity during 2 seconds, then disappear. The displacement of the ship after the bullet has been fired should not influence its trajectory. No bullet can be fired again until the first one has disappeared, as the ship needs to reload its gun.

Hints: In this case, the motion equations are simpler, since the velocity is fixed
The bullet position (bx,by) is initialized once at the position on the ship

```
bx = x0
by = y0
```

and with an initial velocity (bvx,bvy) that is in the forward direction (dx,dy) with fixed speed `bulletVelocity` (in pixel/s).
Also save the time at which the bullet was fired to be able to destroy it after 2 seconds. The model should therefore record both if the bullet is active or not, and if active, the firing time.
The bullet state is updated for each new frame by integrating its own velocity (bvx,bvy)

```
bx = bx + bvx * elapsed
by = by + bvy * elapsed
```

You need to store all the necessary data inside the model to record the state of the bullet. The display of the bullet should be in a separate `drawBullet` function that reads all information from the model to respect MVC separation principles. In particular, the bullet should be drawn only at each tick, not when activating it in the model.

Part 2: Transform Composition [30]

Objective: Use transform composition to draw articulated objects.

Use template code `puppet.html` that contains a canvas and a GUI to modify the parameters.
You need to submit only the final code, as the questions build on each other.

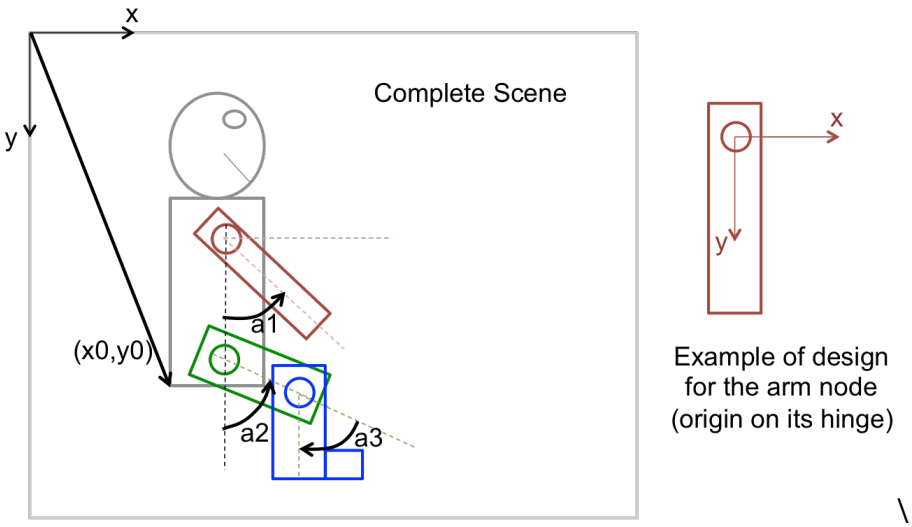

Figure 2: General composition of the puppet scene (left), with detail on the arm node (right).
The scene depends on 5 parameters: position (x0,y0) of the robot, and joint angles a1, a2 and a3.

The objective is to reproduce the scene of Figure 2 using the canvas drawing and transformations.

Q1. Design of scene graph [10]

This question should be answered using diagrams and equations: you may write it on paper and then scan it, or use directly a vector graphics program (powerpoint, illustrator, …). Submit as a PDF.
Draw the scene-graph associated with this scene, and specify the transformation between each node and its parent using a composition of symbolic operators T(dx,dy) and R(angle). The root is the scene coordinate frame, with body as its child.

Q2. Articulated puppet [5]

Create functions that draw each individual part of the puppet in their own coordinates frames:

- `drawArm()  // An example for this part is shown in Figure 2`
- `drawUpperLeg()`
- `drawLowerLeg()`
- `drawBodyAndHead()`

Hints:

To simplify the design, you should design each arm or leg node with the origin located on their hinge.
To test these functions, you may redefine the origin of the canvas as follows and draw only one part. For instance, to draw the arm with its origin at (100,100):

```
ctx.setTransform(1, 0, 0, 1, 0, 0);  // Reset to identity transform
ctx.translate(100,100);              // Change origin to 100,100
drawArm();                           // Draw in arm coordinate frame
```

Q3. Articulated puppet [15]

Create a function `drawPuppet()` that uses the functions defined in Q2 to draw the scene. The translation of the puppet based on parameter (x0,y0) and the articulations based on angles a1, a2

and a3 should be performed using the composition of canvas transforms (`ctx.translate`, `ctx.rotate`…). Erase and redraw the whole scene whenever any parameter changes.

Hints:
To draw the complete puppet, start with the first node (body), then add the first level (arm, upperLeg) and test it, finally add the second level (lowerLeg) and test it. You may use `ctx.save` and `ctx.load` to draw two nodes that depend on the same parent (arm and upperLeg both need the body transform to be applied first).

Bonus work: (worth max 10 points depending on complexity: pick one, or propose something of similar difficulty)
In part1:
- Replicate the asteroid game by adding a few asteroids that can be divided/destroyed by bullets. The asteroid follow a 'constant velocity' model where their velocity vector (vx,vy) remains constant once it has been defined. The asteroids also rotate at constant rotational velocity.

In part2:
- Design a more complex scene with textures, for instance inspired from https://www.pinterest.com/xshopper/articulated-fun/
- Animate your puppet with a realistic complex movement such as walking. *Hint: First define all parameters at a set of keyframes occurring at known times. For a given time elapsed since the start of animation, find the two associated keyframes (the previous one and the next one) and interpolate the parameters with linear interpolation from the parameters at the keyframes.*
- Allow the puppet joints to be manipulated directly by dragging each joint with the mouse. *Hint:* Dragging the body is the easiest, since it is just a translation. dragging an arm or leg is more complex, as it requires estimating the angle from the position of the mouse, since the hinge is fixed: use `Math.atan2()`