서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

# Programming Assignment Report - #1

Yonghun Choi (2019-17862)
yongdori@snu.ac.kr
Seoul National University - Computer Graphics (Spring 2024)

## 1 Instruction

The program in this task implements a simulation of a quadrupedal robot that moves according to the user's input. By implementing the robot's leg joint structure and movement, the program met all the requirements of the task (modeling, animation). The detailed behavior of the program is described below.

- Cursor: The robot looks at your cursor. If you move the cursor to a different location than the direction the robot is looking, the robot will rotate in the direction the cursor is looking.
- Right-click: The robot moves toward the specified location. If the button is held down, the target point is updated to the cursor's position.
- Left-click: The robot fires bullets from its muzzle. Each click fires one bullet alternately left and right.

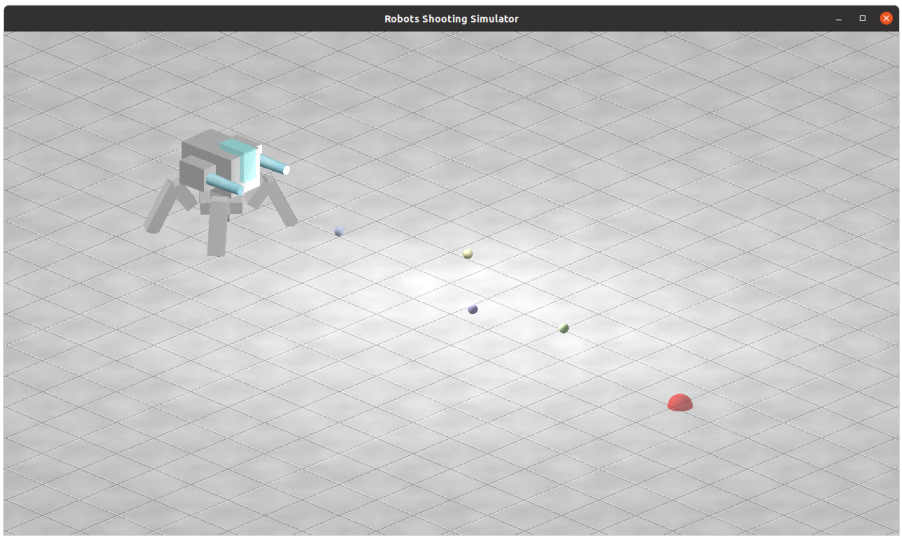


Figure 1: The screen where the program is running.

If you want to see how it works in more detail, check out the attached video or this link.

# 2 Implementation

Here is a brief overview of the steps to implement and render the movement of a quadruped robot.

1. Determine root state : Calculate the coordinates of the goal point corresponding to the cursor position specified by the user. Move the position of the robot based on the target point (input) and the current state (position and velocity). It uses a second-order dynamic system.
2. Determine joint angles : Calculates the appropriate leg angles based on the robot's position. Uses Procedural animation and Inverse Kinematics.
3. Hierarchical Modeling : Each object has a tree structure with the robot's body as the root. From the root, we can recursively calculate the transform matrix.
4. Rendering, Shading : Consider light effects, etc. to make the scene more aesthetically pleasing.

For each of these steps, key techniques are explained in detail below.

## 2.1 Second-order dynamics

To express more dynamic movement of robot, second-order dynamics is used, instead of moving robot to the target position with constant velocity. Relationship between input $x$ and output $y$ follows the equation below.

$$y + k_1\dot{y} + k_2\ddot{y} = x + k_3\dot{x} \tag{1}$$

Using Semi-implicit Euler Method, position cal be calculated iterative as following equation. All next states can be calculated using current state values and user input $x_{n+1}$.[1]

$$\dot{x}_{n+1} = (x_{n+1} - x_n)/T \tag{2}$$
$$y_{n+1} = y_n + T\dot{y}_n \tag{3}$$
$$\dot{y}_{n+1} = \dot{y}_n + T(x_{n+1} + k_3\dot{x}_{n+1} - y_{n+1} - k_1\dot{y}_n)/k_2 \tag{4}$$

Instead of using $k$ directly as an argument for initialization and tuning, the following factors are used. $f$ is the natural frequency, which is related to the response speed of the system. $\zeta$ is the damping coefficient, the larger it is, the more it tends to stay on the target. $r$ affects the initial response.

$$f = \frac{1}{2\pi\sqrt{k_2}},\ \zeta = \frac{k_1}{2\sqrt{k_2}},\ r = \frac{2k_3}{k_1} \tag{5}$$

The code looks like this In actual operation, if the time interval is too large, it may diverge, so in the matrix representation of the above expression, calculate the threshold time for the eigenvalue to be less than 1 in advance, and if the time interval is smaller than that, iterate in smaller intervals.

```
class SecondOrderDynamics:
    def __init__(self, f:float, z:float, r:float, x0:Vec3=Vec3()):
        self._k1 = z/(np.pi*f)
        self._k2 = 1/((2*np.pi*f)**2)
        self._k3 = r*z/(2*np.pi*f)
        self._dt_crit = 0.8 * (np.sqrt(4*self._k2+self._k1**2)-self._k1)
        self._xp = x0 # previous input
        self._y = x0 # output position
        self._yd = Vec() # output velocity

    def update(self, dt:float, x:Vec3):
        xd = (x - self._xp) / dt
        self._xp = x
        it = int(np.ceil(dt / self._dt_crit))
        dt /= it
        for i in range(it):
            self._y = self._y + self._yd * dt
            self._yd = self._yd + (x + xd*self._k3 - self._y - self._yd*self._k1) *
     dt/self._k2
```

## 2.2 Procedural Animation

We set up procedural animations for the robot's four feet. Each foot is grounded to its current position by default. Each foot then targets a position at a specified offset from the body. When this target position is farther away than a predetermined distance, the foot moves toward the target. The speed at which the toe moves is proportional to the robot's movement speed (and rotation speed), and it moves away from the floor as the step progresses. This height follows a custom function, in this case a quadratic function. When the toe reaches the target point, that foot is grounded again, and the cycle repeats.To ensure that the robot moves its feet in a zig-zag pattern, it keeps itself grounded as long as the other foot is not grounded, even if the target location is farther away.[2]

This is the basic principle, and requires further tuning for more natural movement. For example, you can make the offset from root to target vary with velocity instead of being constant, or specify a minimum travel speed for steps, etc. Below is a very brief pseudocode to implement the quadruped animation.

**Algorithm 1** Procedural Animation

```
procedure ProceduralAnimationUpdate(Δt)
    for all leg in legs do
        T ← body.position+leg.offset ▷ Can be set differently depending on the desired motion
        P ← leg.position ▷ Where the toe is projected on the ground
        if leg.grounded then
            if |T − P| > d_crit and neighbor legs are grounded then
                leg.grounded ← False
                leg.target ← T
            else
                leg.position ← leg.position + calcV(T, P, ...)Δt
                leg.end_point ← leg.position + calcH(T, P, ...) ▷ Used for IK
                if |T − P| < ϵ then ▷ Very close to target
                    leg.grounded ← True
                    leg.position ← leg.target
                    leg.end_point ← leg.target
```

## 2.3 Inverse Kinematics

Now that we have the position of the root and the position of the end-point, we can use Inverse Kinematics to calculate the angle of each joint. Using the initial joint position of the root(body) as the origin, this can be interpreted as the 2-link kinematics of planar robotic arm.

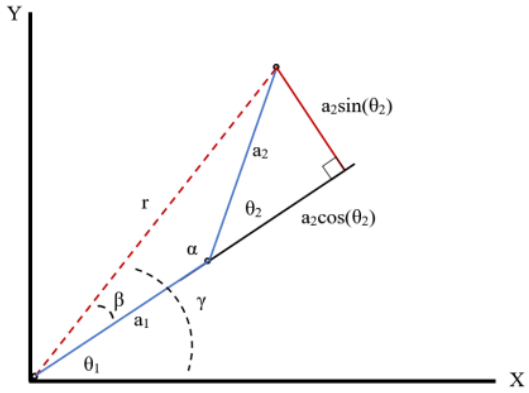


Figure 2: 2-Link Kinematics of Planar Robotic Arm.

This yields the only unambiguous solution as follows.[3]

$$\theta_2 = \pm \arccos\left(\frac{x^2+y^2-{a_1}^2-{a_2}^2}{2a_1a_2}\right) \tag{6}$$

$$\theta_1 = \arctan\left(\frac{y}{x}\right) \mp \arctan\left(\frac{a_2 \sin\theta_2}{a_1+a_2 cos\theta_2}\right) \tag{7}$$

## 2.4 Hierarchical Modeling

The robot has a hierarchy with the body as the root. Each object (legs and arms) has a local coordinate relative to its own joint. The object stores the translation and rotation of its joints from the parent's coordinates as a matrix. Then, when root recursively calls transform matrix calculation and receives the result from the parent, it can calculate the transform matrix of all objects. The calculated transform matrix is then passed as uniform to the object's shader program on every update to be used for rendering.

**Algorithm 2** Transform matrix calculation of hierarchical modeling

```
procedure CALCULATETRANSFORM(obj) ▷ call CALCULATETRANSFORM(root) every update
    T ← obj.translate_mat
    R ← obj.rotation_mat
    T_par ← obj.parent.transform_mat
    obj.transform_mat ← T_par T R
    for all child ∈ obj.children do CALCULATETRANSFORM(child)
```

## 2.5 Lighting

Taking real-world lighting into account is a very difficult and complex task, but here we use a very simplified model that doesn't even consider shadows. The lighting model consists of three elements

- Ambient: Takes into account the effect of lights that illuminate the entire world. Because of this, objects are never completely dark. It acts evenly on all surfaces.
- Diffuse: Consider the effect of directional light. This models the fact that light receives more energy when it is incident perpendicular to a surface.
- Specular: This primarily reflects the effect caused by dot light, which tracks light bouncing off a surface. Glowing highlights are an example of this.

Ambient light is applied equally to all surfaces, so it can simply be represented as a constant.

Diffuse should be larger the more perpendicular the surface is to the direction of light. This can be obtained by cross-producing the direction of light with normal. The direction of the light can be found by finding the direction vector from the surface to the light for dot light, or by changing the sign for directional light.

Specular needs to calculate how close the direction of the reflected light is to the camera. If the direction of the reflected light from the surface and the direction of the camera coincide, the light direction vector and the camera direction vector together will be parallel to the normal. Therefore, the light direction vector and the camera direction vector are combined, normalized, and cross-producted with the normal. Since it is natural for highlights to brighten sharply when the reflection angle matches, we fit an exponential function.[4]

$$\mathbf{d}_{light} = \text{normalized}\ (\mathbf{x}_{light} - \mathbf{x}_{vertex}) \tag{8}$$

$$\mathbf{d}_{cam} = \text{normalized}\ (\mathbf{x}_{cam} - \mathbf{x}_{vertex}) \tag{9}$$

$$\mathbf{h} = \text{normalized}\ (\mathbf{d}_{light} + \mathbf{d}_{cam}) \tag{10}$$

$$L = A + I(\mathbf{n} \cdot \mathbf{d}_{light}) + (\mathbf{n} \cdot \mathbf{h})^S \tag{11}$$

Adding all of these values together gives us the brightness, which we multiply by the vertex's color and pass to the fragment. In code, this would look like this The parts that are not related to lighting have been omitted.

```
layout(location =0) in vec3 a_vertex;
layout(location =1) in vec3 a_normal;

out float v_light;

uniform vec3 u_lightPos;
uniform vec3 u_camPos;

uniform mat4 u_transform;
uniform mat4 u_viewProj;

uniform float ambient = 0.2f;
uniform float intensity = 0.5f;
uniform float shininess = 100.0f;

void main()
{
    vec4 worldPos = u_transform * vec4(a_vertex, 1.0f);
    gl_Position = u_viewProj * worldPos; // local -> world -> vp

    vec3 normal = (u_transform * vec4(a_normal, 0.0f)).xyz; // world coord normal
    vec3 lightDir = normalize(u_lightPos - worldPos.xyz);
    vec3 camDir = normalize(u_camPos - worldPos.xyz);
    vec3 half = normalize(lightDir + camDir);

    float diffuse = max(dot(normal, lightDir), 0.0f);
    float specular = pow(max(dot(normal, half), 0.0f), shininess);

    v_light = ambient + diffuse*intensity + specular;
}
```

# References

[1] t3ssel8r. Giving personality to procedural animations using math, 06 2022.

[2] Codeer. Unity procedural animation tutorial (10 steps), 03 2020.

[3] Norberto Torres-Reyes. 2-link-kinematics [dasl wiki], 09 2018.

[4] gfx fundamentals. Webgl 3d - point lighting.