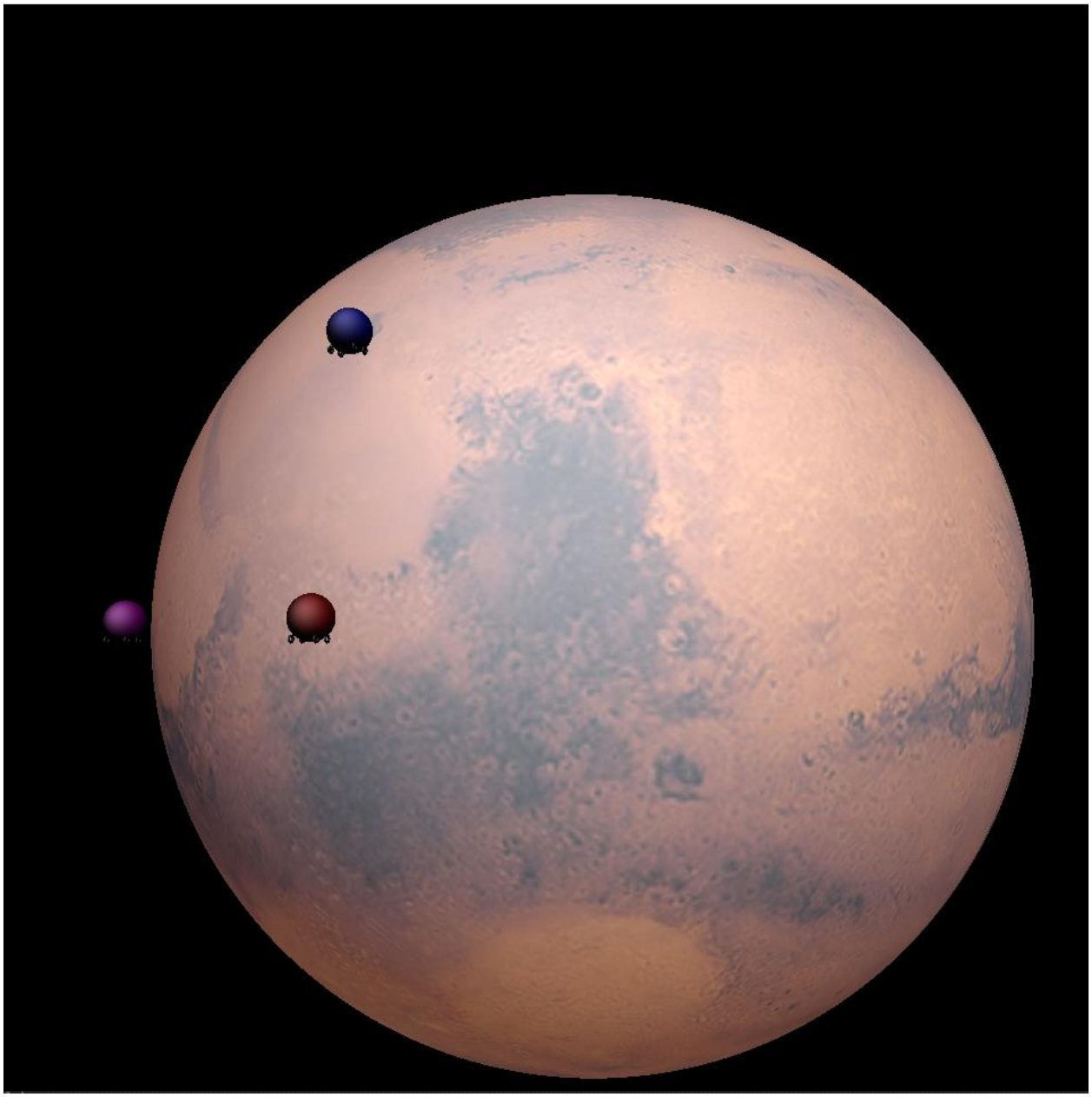
Efehan Güner 25151

CS 405 3D Project Part 2

1)



Mars is modeled as a sphere. It has a texture mapped to it coming from an asset named “mars_1k_color.jpg” which is surface of Mars in an image format. Texture is mapped on Mars using these parameters in OpenGL:

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);


    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
```

For shading Blinn-Phong shading is used with a directional light from coordinates: (-1, -1, 1).

Shading blending is achieved in the shader using surface uv's of the sphere and using them in the texture() function of OpenGL. You can see the fragment shader below:

```
void main()
{
    vec3 color = vec3(0);

    vec3 surface_position = world_space_position.xyz;
    vec3 surface_normal = normalize(world_space_normal);
    vec2 surface_uv = vertex_uv;
    vec3 texture_color = texture(u_texture, surface_uv).rgb;
    texture_color=texture_color * textured;
    vec3 surface_color = u_surface_color;

    vec3 ambient_color = vec3(0.7);
    color += ambient_color * surface_color * texture_color;

    vec3 light_direction = normalize(vec3(-1, -1, 1));
    vec3 to_light = -light_direction;

    vec3 light_color = vec3(0.3);

    float diffuse_intensity = max(0, dot(to_light, surface_normal));
    color += diffuse_intensity * light_color * surface_color;

    vec3 view_dir = vec3(0, 0, -1);
    vec3 halfway_dir = normalize(view_dir + to_light);
    float shininess = 4;
    float specular_intensity = max(0, dot(halfway_dir, surface_normal));
    color += pow(specular_intensity, shininess) * light_color;

    out_color = vec4(color,1);
}
        )FRAGMENT");
```

2)

Firstly, perspective projection is applied to the scene to allow camera control realistically. There is need for two controls: Mouse control is used to change look at coordinates of the camera, WASD keys are used to move position of the camera itself. In the mouse control, there is a sensitivity set to control how much look at coordinates change when mouse position changes. This sensitivity is then multiplied with present_mouse_position last_mouse_position. Finally camera coordinates are calculated using two variables named pitch and yaw using some trigonometry:

```
    float sensitivity = 0.2f;  //0.1f
    xoffset *= sensitivity;
    yoffset *= sensitivity;

    yaw += xoffset;
    pitch += yoffset;

    if (pitch > 89.0f)
        pitch = 89.0f;
    if (pitch < -89.0f)
        pitch = -89.0f;

    glm::vec3 direction;
    direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    direction.y = sin(glm::radians(pitch));
    direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    camera_front = glm::normalize(direction);
```

For controlling the camera position wasd is used w and s for the z axis; a and d for the x axis. Camera speed variable is used to control the speed of camera multiplying it with the camera front. They are then equated to the camera position:

```
if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {
    if (camera_mode)
        camera_position += cameraSpeed * camera_front;
    if (rover_mode)
        //movement += rover_speed * glm::vec3(0, 0, 0.1f);
    {
        movement += delta_time * glm::vec3(0, 0, rover_speed);
        rotate_tires1 = true;
    }

}
if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
    if (camera_mode)
        camera_position -= cameraSpeed * camera_front;
    if (rover_mode)
        //movement += rover_speed * glm::vec3(0, 0, -0.1f);
    {
        movement += delta_time * glm::vec3(0, 0, -rover_speed);
        rotate_tires1 = true;
    }
}
if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS) {
    if (camera_mode)
        camera_position += glm::normalize(glm::cross(camera_front, camera_up)) * cameraSpeed;
    if (rover_mode)
        //movement += rover_speed * glm::vec3(-0.1f, 0, 0);
    {
        movement += delta_time * glm::vec3(-rover_speed, 0, 0);
        rotate_tires1 = true;
    }
}
if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS) {
    if (camera_mode)
        camera_position -= glm::normalize(glm::cross(camera_front, camera_up)) * cameraSpeed;
    if (rover_mode)
        //movement += rover_speed * glm::vec3(0.1f, 0, 0);
    {
        movement += delta_time * glm::vec3(rover_speed, 0, 0);
        rotate_tires1 = true;
    }
```

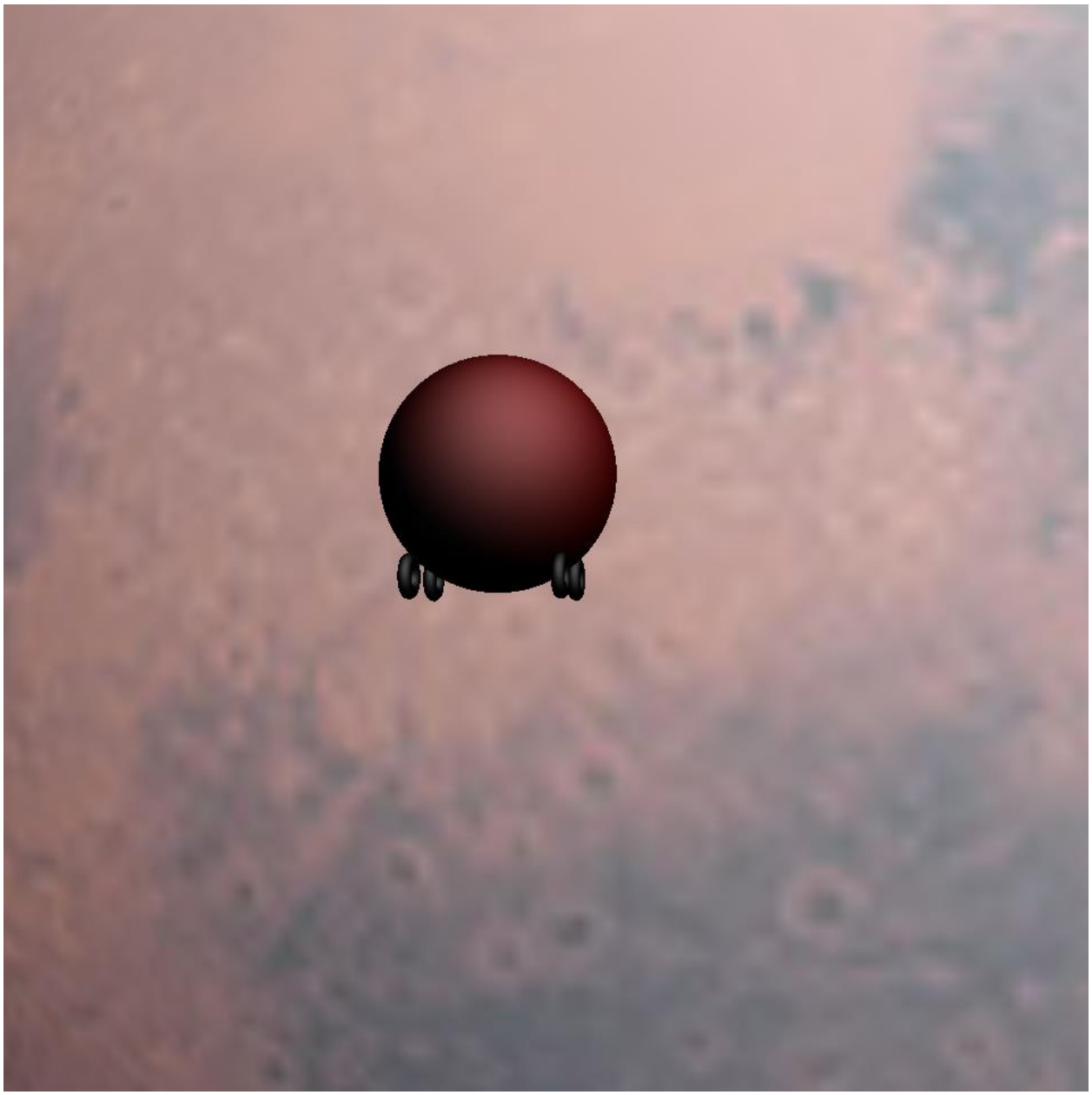
Finally camera is calculated using glm's lookat() function:

```
auto view = glm::lookAt(
    camera_position,  //glm::vec3(0, 0, -1.15)
    camera_position + camera_front,
    camera_up
);
auto projection = glm::perspective(glm::radians(45.f), 1.f, 0.1f, 10.f); //far was 10.f

glUniformMatrix4fv(projection_view_location, 1, GL_FALSE, glm::value_ptr(projection * view));
```

*Press C at the start to move the camera.

3)

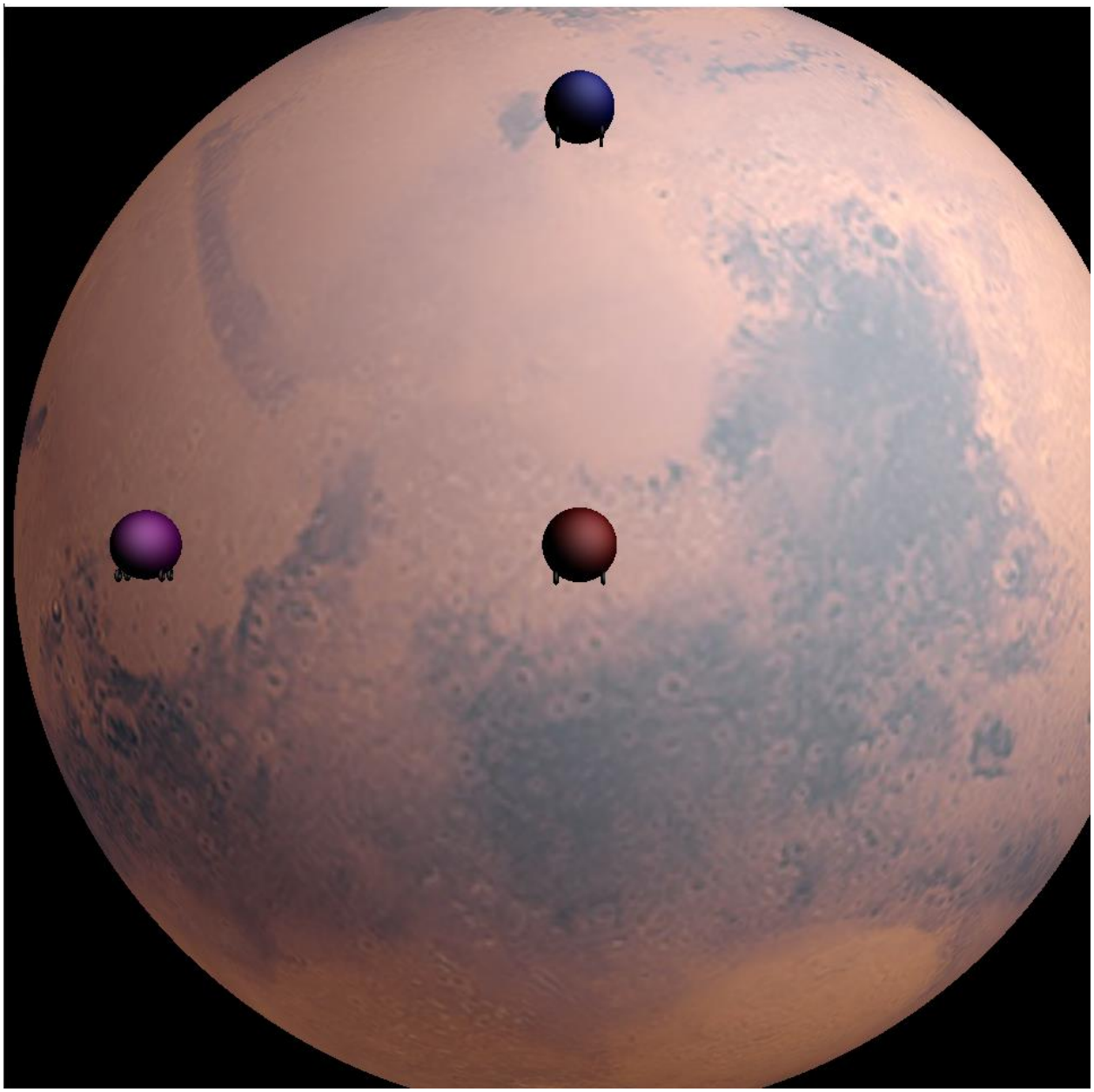


There are two modes now (Initially there is no mode in the program you have to press C or R at the beginning to control rover or the camera.): One camera mode activated by pressing “C” key and rover mode activated by pressing ““R” key to control the rover by pressing WASD. Mouse always controls the lookat positon of camera in both modes. WASD keys work as in same logic with the camera mode allowing rover to change positon in the space freely. There is a rover speed variable to control the displacement of the rover per delta time (movement=delta time * directional speed(rover_speed)). Than this newly achived movement variable (distance travelled) is used by translating in and joining it with the transformation matrice. Same transformation matrice is used for the 4 wheels except the scale of the rover’s body. Wheels also spin using some rotation calculations only when the rover moves. Rover is

constructed from one large sphere and 4 torus's which can rotate if there is movement.(Pay attention to see the movement it may be subtle in some viewing angles.)

*I couldn't move rovers on the surface(I tried it for several days but no luck), so they move freely in the space. This is because angular displacement calculations was extremely difficult resulting with buggy (kind of teleporting) movement. So it was scrapped for the free motion in space(linear motion).

4)



Two additional rovers are constructed using same transformation matrice with some translation offsets.

Other rovers(Blue and Purple) will always move until they catch the user rover (collide with it). Their position is calculated using glm's mix function which achives interpolation between user rover and the chasing rover. Purple rover is faster than the blue rover which is achieved by giving 0.98 to the mix fuction of the purple instead of 0.99. They both chase the red user rover till they collide with it

For collison, AABB type collision is used using an cube shaped collision detector. We model this invisible cube using radius of the rover's main body. Collison detection function is as follows:

```
bool CheckCollision(glm::vec3& first, glm::vec3& second) { // AABB - AABB collision

    auto r = 0.0763892f; //0.0723892f      //radius of sphere
    // collision x-axis?
    bool collisionX = first.x + r >= second.x - r &&
        second.x + r >= first.x - r;
    // collision y-axis?
    bool collisionY = first.y + r >= second.y - r &&
        second.y + r >= first.y - r;
    // collision z-axis?
    bool collisionZ = first.z + r >= second.z - r &&
        second.z + r >= first.z - r;
    // collision only if on all axes
    return collisionX && collisionY && collisionZ;
}
```

When collison is detected between blue and red sphere or purple and red sphere, red user rover stops, never to move again if program is not restarted. Consequently chasing rovers stop because they caught the user rover. Camera mode still works in this situation.

*For debugging purposes there should be console outputs in case of a collision.

Collision (Between purple and red):