Real-Time Polygon Mesh Deformation

Alessio Quercia - 918925

Abstract

The goal of this project was to create an application that shows 3D real-time polygon mesh deformation on collision between two objects. The objective was achieved by using two different approaches: the first one uses arrays of uniforms in the vertex shader to store the hit points and directions, whereas the second one uses the feedback transform technique in a first render pass, updates the meshes and then renders normally at a second pass. The second method resulted to be more efficient and allowed an infinite number of deformations in the scene.

1 Introduction

OpenGL is a specification, developed by Khronos Group, that provides a set of functions useful to manipulate graphics. It specifies what the output of each function should be and how it should perform, leaving freedom to developers on how to implement it. OpenGL offers two modes nowadays: immediate mode and core-profile mode. The first one is also called fixed function pipeline and it is mostly composed by non-programmable elements, which offers no control on the order of the operations to apply and gives less freedom to the developers on the programming parts. The second one, on the contrary, is more flexible and offers more programmable parts (shaders) and control on the order of operations. It is available from OpenGL 3.3, which is the version adopted to develop this project [1].

The goal of this project was to create an application that allows to deform the polygon mesh of 3D objects by shooting them in real-time. In order to handle the physics (e.g. the colliders and the collisions) in real-time inside the application, Bullet library has been adopted [2]. Moreover, a specific function in the vertex shader has been used to apply the deformations on the vertices on impact.

Shaders are programs running on the GPU, each one having a specific task. The language adopted by the shaders is GLSL. There are four main shaders: vertex shader, texellation shader, geometry shader and fragment shader. The peculiarity of these shaders is that they pass their outputs as inputs to the next shader. The first shader is the vertex shader, which manipulates vertices one by one. It is possible to apply any function on the vertices, but it is not possible to add or remove them. The second one is the tessellation shader, which allows to render curved surfaces (such as Beziers surfaces); the third one is the geometry shader, in which the geometry can be manipulated, e.g. the triangles of a mesh. In this shader it is possible to add or even remove vertices. Both tessellation and geometry shaders are optional. The last one is the fragment shader, which, after a rasterization stage in which the fragments are computed interpolating vertices, it is possible to directly act on the single fragments and to assign them a color that will be displayed on the screen.

Thanks to a proper use of the vertex, the geometry and the fragment shaders, it was possible to achieve the goal, as described in the following sections.

2 Method

Different approaches can be used to deform a polygon mesh in real-time. In this project two approaches have been tested and compared. Both the methods use Bullet library to detect the collision points, a deformation function (section 2.3) in the vertex shader to apply the mesh deformations in real-time, and some light functions (section 2.5) to illuminate the scene.

2.1 Uniforms array approach

The uniform array method consists in storing the last N hit points and hit directions in two distinct arrays of uniforms in the vertex shader (shown as pseudo-code in Algorithm 4). The points and the directions are then used to compute the deformations to the vertices which are within a specific range. Therefore, each time a deformable object is hit by a projectile, the corresponding hit point and projectile direction are stored in two distinct arrays (hit points and hit directions). One rendering pass is enough to render the whole scene, given that each object is rendered by a vertex shader, which (other than computing the Model, View and Projection transformations, as explained in Section 2.4) receives both the described arrays, apply the vertices deformations (if any) to the object mesh, and by a fragment shader (shown as pseudo-code in

Alessio Quercia, Real-Time Graphics Programming, A/A 2018-2019, Università degli Studi di Milano, via Celoria 18, Milano, Italy
E-mail: alessio.quercia@studenti.unimi.it

Algorithm 7), which applies the texture (if any) and the illumination models. The method (of which pseudo-code is shown in Algorithm 1) proceeds as follows:

- Constantly check for new collision points in the main loop.
- If a new collision point is detected:
  - Its world coordinates are stored in a list of hit points.
  - The direction with which the projectile hit the object is stored in a list of hit directions.
- The last N hit points and directions stored so far are fed to the vertex shader, which, for each vertex, computes:
  - The vertex world coordinates.
  - The distance between the vertex world position from every hit point in the list and, if the distance is smaller than a certain constant, it applies the deformation on that vertex.
  - The view direction, normal and the vertex projection coordinates.
- On the fragment shader, for each fragment, the color is computed as the sum of the lights coming from different sources: directional light, point lights and spotlight. Any of these lights can be turned on or off.

Algorithm 1 Uniforms array method

```
Initialize variables
while not exit do
  Update game logic
  if new collision detected then
    Add hit point to hit points (if limit is reached, discard the oldest point and add the new one)
    Add hit direction to hit directions (if limit is reached, discard the oldest point and add the new one)
  Feed model, view, projection transforms, hit points and hit directions to the vertex shader
  Feed a texture (optional) and the illumination model variables to the fragment shader
  Render scene
```

The main problem of this approach is that all the N deformations in the scene have to be computed at each frame, that is each model has to loop over all the N possible points and apply the deformations that belong to it. This results to be very slow when loading lots of models and models with a high number of vertices.

2.2 Feedback transform approach

The feedback transform method is based on the deferred rendering technique, which uses multiple rendering passes in a single render step, where only the last rendering pass is going through the fragment shader and therefore applying the illumination models, light effects and actually computing the fragments color. All the previous rendering passes are usually used to update the geometry of a given object. In general, the feedback transform approach consists in ”capturing” some variables in the vertex or in the geometry shader and in sending them back to the main loop, without passing them to the fragment shader [3], [4]. This can be useful when some variables need to be processed multiple times before being sent to the fragment shader.

In this case, two rendering passes have been used: the first one computes the deformations on the models hit by a projectile, whereas the second one ”normally” renders the scene. During the first rendering pass, a vertex shader (shown as pseudo-code in Algorithm 3) specialized in applying the deformations to the vertices positions and normals is used. For each vertex of an hit object, this vertex shader receives as input its position, its normal, the hit point and its corresponding hit direction, and it applies a deformation function (described in Section 6). The vertex shader ”captures” two variables at the end: the new position (brought back in local coordinates by multiplying the inverse of the model matrix and the vertex position in world coordinates) and the new normal. Even if a vertex position and normal are not modified by the deformation, they are ”captured” and sent back to the main loop in order to have an array of the same shape as the input array (containing all the object vertices positions and normals). Indeed, after the first rendering pass, an array containing the updated vertices positions and normals is returned for each hit object to the main loop. Therefore it is possible to update the hit objects meshes by replacing their vertices positions and normals (only the ones that have actually been modified) with the deformed vertices positions and normals. Once the hit object meshes have been updated, the second rendering pass can be applied. In this pass the scene is ”normally” rendered, by using a different vertex shader (shown as pseudo-code in Algorithm 5), which computes (for each object) the model, view and projection transformations (see Section 2.4), and a fragment shader (shown as pseudo-code in Algorithm 7), which finally applies the illumination model (see Section 2.5). The method (of which pseudo-code is shown in Algorithm 2) proceeds as follows:

- Constantly check for new collision points in the main loop.
- If a new collision point is detected:

  - The hit point is fed to a first vertex shader which computes the deformation for any vertex (of the hit object) with a distance to the fed point which is smaller than a certain constant, and then returns to the main loop the list of the (hit object) new vertices positions and normals.
  - The hit object mesh is updated with the new vertices positions and normals.

- The object is rendered normally:
  - On the vertex shader, for each vertex, the world position, normal and texture coordinates are computed and sent to the fragment shader.
  - On the fragment shader, for each fragment, the color is computed as the sum of the lights coming from different sources: directional light, point lights and spotlight. Any of these lights can be turned on or off.

Algorithm 2 Feedback transform method

```
Initialize variables
while not exit do
  Update game logic
  if new collision detected then
    Feed model, view, projection transforms, the hit point and the hit direction to a first "deform" vertex shader
    Render the hit object and stop before the fragment shader stage, by capturing the new vertices and normals in the vertex shader
    Update the hit object mesh with the deformed vertices and normals
  Feed model, view, projection transforms to a second "render" vertex shader
  Feed a texture (optional) and the illumination model variables to the fragment shader
  Render scene
```

In this approach, with respect to the other one, only the hit model is updated, by capturing in a first vertex shader the vertices and the normals, and by updating the mesh only if the vertices/normals have been modified. This method results to be much more efficient than the other one and allows to have an infinite number of deformations in the scene.

2.3 Deformation function

The deformation function (shown in Algorithm 6) is called inside the vertex shader (therefore it is called for each vertex of a mesh) and, given as input an hit point, the corresponding hit direction, a vertex position and normal, it produces as output the corresponding deformed vertex position and normal. The deformation is only computed if the Euclidean distance between the vertex position and the hit point is smaller than a given max distance (which represents the radius within which a deformation is having effect, given the hit point as center). In that case, the vertex position and normal are translated as follows:

$$new\_position = position + direction \tag{1}$$

$$new\_normal = normalize(normal + direction) \tag{2}$$

where:

$$direction = hitDirection \times magnitude \tag{3}$$

$$magnitude = min\left(\frac{power}{distance}, max\_magnitude\right) \tag{4}$$

$power$ is a constant, $distance$ is the Euclidean distance between the vertex position and the hit point, and $max\_magnitude$ is a constant preventing the vertex position and normal to be deformed too much. The ratio $\frac{power}{distance}$ guarantees that the magnitude of a deformation decreases with the increase of the distance from the hit point. Therefore the closer a vertex position (and corresponding normal) is to an hit point the more it will be affected by the deformation (and viceversa), having as a limit the constant $max\_magnitude$. Different values have been tested for the constants, and a quite good result has been obtained by setting $max\_distance = 0.5$, $power = 0.05$ and $max\_magnitude = 0.3$. By increasing the $power$ (and the $max\_magnitude$) it is possible to have deeper deformations (and viceversa), whereas by increasing the $max\_distance$ it is possible to have larger deformations (and viceversa).

Algorithm 3 Deformation vertex shader

Input: hitPoint, hitDirection, localPosition, normal, model
Initialize constants (max_distance, power, max_magnitude)
newPosition = localPosition
newNormal = normal
worldPosition = model · localPosition
distance = Euclidean_distance(hitPoint, worldPosition)
if distance < max_distance **then**
  newPosition, newNormal = Deform(hitDirection, worldPosition, normal)
  newPosition = inverse(model) · newPosition
Output: newPosition, newNormal

Algorithm 4 Rendering vertex shader (Uniforms Array Approach)

Input: position, normal, texCoords, hitPoints, hitDirections, model, view, projection
Initialize constants (max_distance, power, max_magnitude)
worldPosition = model · localPosition
for hitPoint in hitPoints **do**
  distance = Euclidean_distance(hitPoint, worldPosition)
  if distance < max_distance **then**
    newPosition, newNormal = Deform(hitDirection, worldPosition, normal)
Normal = transpose(inverse(model)) · normal
mvPosition = view · worldPosition
viewPosition = −mvPosition
gl_Position = projection · mvPosition
Output: viewPosition, worldPosition, Normal, texCoords

Algorithm 5 Rendering vertex shader (Feedback Transform Approach)

Input: position, normal, texCoords, model, view, projection
worldPosition = model · localPosition
Normal = transpose(inverse(model)) · normal
mvPosition = view · worldPosition
viewPosition = −mvPosition
gl_Position = projection · mvPosition
Output: viewPosition, worldPosition, Normal, texCoords

Algorithm 6 Deform

Input: hitDirection, worldPosition, normal
magnitude = $power/distance$
magnitude = min(magnitude, max_magnitude)
direction = $hitDirection \times magnitude$
newNormal = normalize(normal + direction)
newPosition = worldPosition + direction
return newPosition, newNormal

Algorithm 7 Fragment shader

Input: viewPosition, worldPosition, Normal, texCoords, Texture (optional), pointLights (array), dirLigth, spotLight
normal = normalize(Normal)
viewDir = normalize(viewPosition - worldPosition)
color = texture(Texture, texCoords)
if dirLight is on **then**
  color + = CalcDirLight(dirLight, normal, viewDir)
for pointLight in pointLights **do**
  if pointLight is on **then**
    color + = CalcPointLight(pointLight, normal, worldPosition, viewDir)
if spotLight is on **then**
  color + = CalcSpotLight(spotLight, normal, worldPosition, viewDir)
Output: color

2.4 Model, view and projection transformations

Model transformation brings an object from its Local reference system to the World reference system, View transformation brings an object from the World reference system to the View (or Camera) reference system, and Projection transformation brings an object from the View reference system to the Projection reference system. After applying all these transformations (in the explained order), the whole scene is represented in the canonical view (half) cube, where it is easier to apply clipping, culling and other post-processing operations. Therefore, the output of a vertex shader will be a position in Projetion space, computed as:

$$p' = M_{proj} \cdot M_{view} \cdot M_{model} \cdot p \tag{5}$$

where M_{model} is a combination of scaling, rotation and translation transformations, M_{view} translates each object to the origin (by E_x, E_y and E_z) and then rotate it so as its World axes (xyz) match the Camera axes (uvw):

$$M_{view} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and M_{proj} is computed as a Shear transformation, given that the a Perspective Projection is computed:

$$M_{proj} = \begin{bmatrix} \frac{1}{ftan(\frac{\theta_h}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{ftan(\frac{\theta_v}{2})} & 0 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where f is the far plane (that is the last visible part from the camera), θ_h is the horizontal field of view (that is the horizontal view angle) and θ_v is the vertical field of view (that is the vertical view angle).

Moreover, to avoid the normals towards each others after having applied a scaling transformation to an object, it is necessary to invert the scaling transformation. To do so, it is required to compute the inverse of the transpose of the Model transformation matrix:

$$normal = (M_{model}^{-1})^T \cdot normal \tag{6}$$

2.5 Light calculation

The lights are computed as suggested in the OpenGL tutorial [1], that is by using three functions, one for each different light source: directional light, point light and spotlight. Each one of these is computed as the sum of ambient, diffuse and specular lighting:

$$fragment\_color = ambient + diffuse + specular \tag{7}$$

In every case, the diffuse lighting is multiplied by a diffuse shading computed as follows:

$$diff = max(normal \cdot lightDir, 0) \tag{8}$$

where $normal$ is the fragment normal and $lightDir$ is the light direction; while the specular light is multiplied by a specular shading computed as follows:

$$spec = (max(viewDir \cdot reflectDir, 0))^{material\_shininess} \tag{9}$$

where $viewDir$ is the view direction, computed as simple difference between the view position and the fragment position, and $reflectDir$ is the reflection direction, which is computed as a reflection of the light direction around the normal.

2.5.1 Directional light

The directional light is given by equations described above, therefore the diffuse lighting is computed as:

$$diffuse = diffuse * diff \tag{10}$$

and the specular lighting as:

$$specular = specular * spec \tag{11}$$

2.5.2 Point light

Point lights are computed in a slightly different way with respect to the directional light. Indeed, starting from the equations of the directional light, a point light is obtained by also multiplying by an attenuation factor the ambient, diffuse and specular lighting. The attenuation is computed as follows:

$$attenuation = \frac{1}{c + linear * distance + quadratic * distance^2} \tag{12}$$

where c (often set to 1) is a constant used to prevent the denominator to become smaller than the numerator (which would allow the light to become brighter), linear is a constant which reduces the intensity of the light in a linear fashion, and quadratic is a constant which reduces the intensity of the light in a quadratic fashion. The quadratic decrease will get larger than the linear one when the distance grows. The distance is computed as follows:

$$distance = ||lightPos - fragPos|| \tag{13}$$

that is, it is the length of the vector given by the difference between the light position ($lightPos$) and the fragment position ($fragPos$).

2.5.3 Spotlight

The spotlight is computed by multiplying the diffuse and specular lighting described in the point lights by an intensity factor, computed as follows:

$$intensity = clamp\left(\frac{\theta - outerCutOff}{\phi}, 0, 1\right) \tag{14}$$

where $clamp$ is a function which restricts the first argument between the values of the second and third arguments, θ is the angle between the light direction and the direction the spotlight is aiming at, and ϕ is the cutoff angle that specifies the spotlight's radius. Everything outside this angle is not lit by the spotlight [1]. Indeed if θ is greater than the $outerCutOff$, attenuation and intesity are multiplied to the diffuse and specular components, in such a way to have light attenuation and intensity decrease outside the spotlight cone.

3 Results

High polygon meshes could be loaded to achieve a better precision during the vertices deformations. Both the approaches have been tested by loading low-poly spheres with 382 vertices (figures 3 and 7), high-poly spheres with 7938 vertices (figures 4 and 8), low-poly cubes with 386 vertices (figures 1 and 5) and high-poly cubes with 6146 vertices (figures 2 and 6). All of which have been loaded without textures, with 1K textures, 2K textures and 4K textures to see the differences in the performance.

In the first approach, the number of total possible vertices deformations loaded at the same time had to be limited to 600, in order to avoid overloading the vertex shader with a number of default vertex uniforms exceeding the hardware limits. Once the limit is reached, the oldest vertices are replaced by the new ones, in such a way to preserve the possibility to deform new vertices. In the second approach, there is no limitation on the number of possible vertices deformations, and the meshes are actually updated in real-time when they are deformed. This means that a vertex that has been deformed doesn't have to be deformed again at the next render step, which is different from the first approach, where at each render step all the N possible vertices have to be deformed.

The results obtained by the first approach are shown in TABLE 1, whereas those obtained by the feedback transform method are shown in TABLE 2. For both the methods frames per second (FPS) have been used as a measure of the performance. The uniforms array approach is clearly inferior in performance with respect to the feedback transform approach. In this last case, the FPS are variable because multiple sequential hits can happen in a second, which implies multiple sequential mesh updates, which require more time to process. The FPS in both cases also depend on the number of projectiles remaining on the ground, given that they need to be rendered at each step too. This issue can easily be solved by removing the projectiles from the scene after a chosen amount of time they have been shot. By doing so, the FPS would be more stable in both cases (especially in the second approach). However, this was not done in this case for it was not the main goal of the project and for time reasons.

3.1 System architecture

I ran the simulations on two laptops, characterized by:

1) Intel Core i7-6700HQ CPU, Nvidia Geforce GTX 950M GPU and 8 GB RAM.
2) Intel Core i7-9750HQ CPU, Nvidia Geforce GTX 1650 GPU and 16 GB RAM.

LP Cubes	HP Cubes	LP Spheres	HP Spheres	FPS L1	FPS L2
3	3	0	10	15	60
3	3	5	5	20	90
3	3	10	0	45	170
6	0	0	10	20	70
6	0	5	5	30	120
6	0	10	0	120	200-350
0	6	0	10	10	50
0	6	5	5	15	70
0	6	10	0	30	110

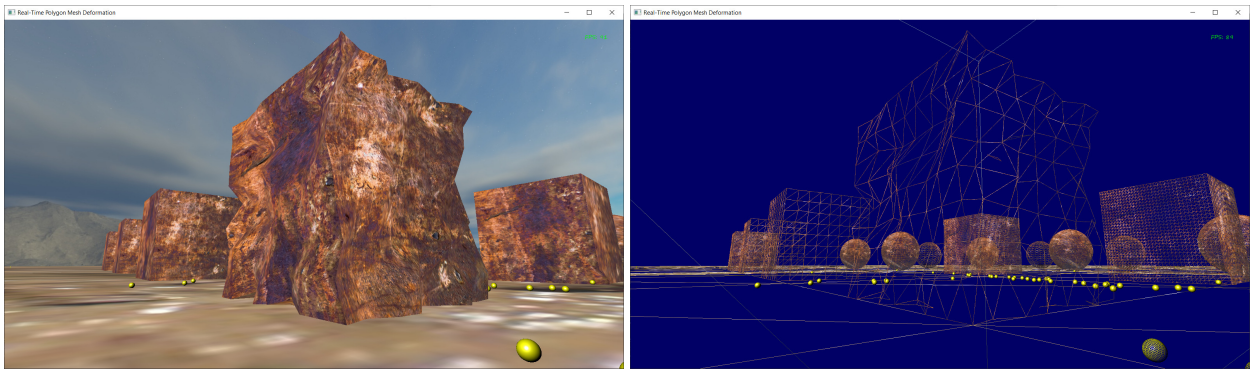
TABLE 1: Uniforms array approach results. LP/HP Cubes/Spheres indicate the number of low-poly/high-poly cubes/spheres in the scene during the test (which is depicted by a row). L1 refers to Laptop 1 and L2 to Laptop 2.

LP Cubes	HP Cubes	LP Spheres	HP Spheres	FPS L1	FPS L2
3	3	0	10	170-270	180-550
3	3	5	5	140-300	180-740
3	3	10	0	160-320	200-780
6	0	0	10	110-290	120-630
6	0	5	5	130-310	150-700
6	0	10	0	140-340	150-760
0	6	0	10	130-260	130-560
0	6	5	5	130-310	140-630
0	6	10	0	140-310	160-700

TABLE 2: Feedback transform approach results. LP/HP Cubes/Spheres indicate the number of low-poly/high-poly cubes/spheres in the scene during the test (which is depicted by a row). L1 refers to Laptop 1 and L2 to Laptop 2.

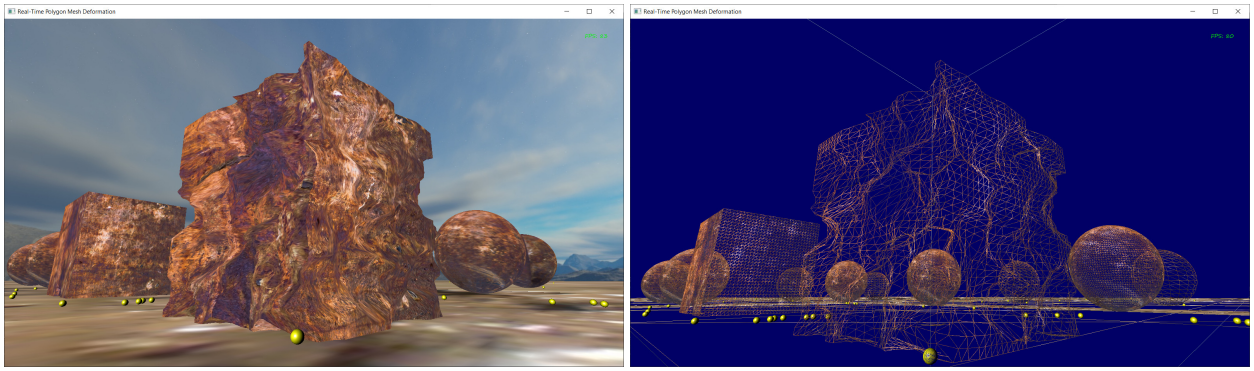
4 Conclusions

The proposed goal was achieved by creating an application in which it is possible to shoot and hit some 3D simple objects (cubes and spheres) by modifying their polygon meshes on impact. Two approaches have been tested: uniforms array and feedback transform. This last one revealed to be much more efficient than the first one, which is also worse because it only allows a fixed number of deformations in the scene, whereas the feedback transform approach allows infinite deformations. It is also possible to extend the application in order to hit and deform 3D objects having more complex shapes, by creating colliders with Bullet library which can fit them.



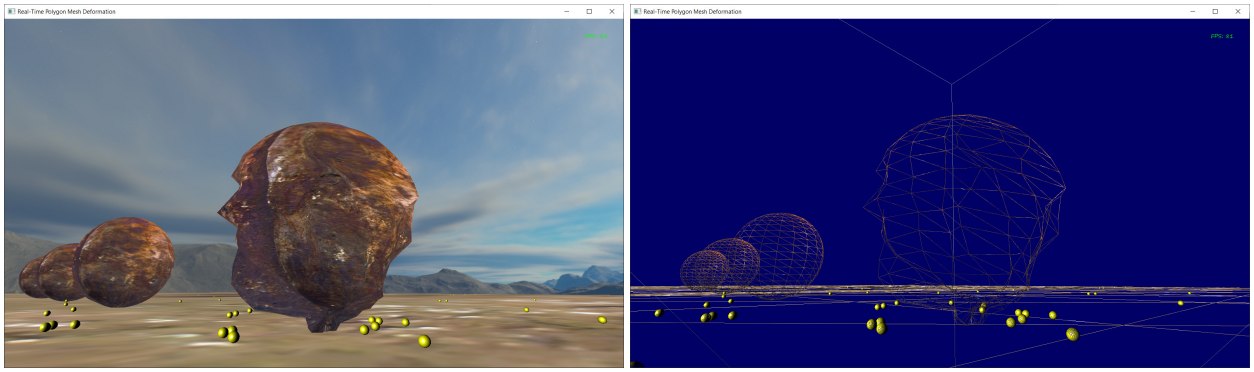
(a) Normal mode. (b) Wireframe mode.

Fig. 1: Deformed cube having low poly mesh obtained using the uniforms array approach.



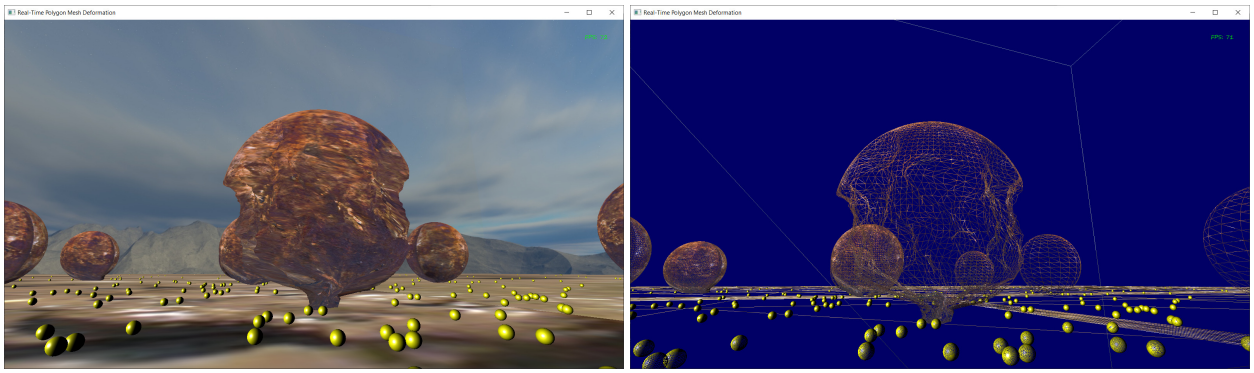
(a) Normal mode. (b) Wireframe mode.

Fig. 2: Deformed cube having high poly mesh obtained using the uniforms array approach.



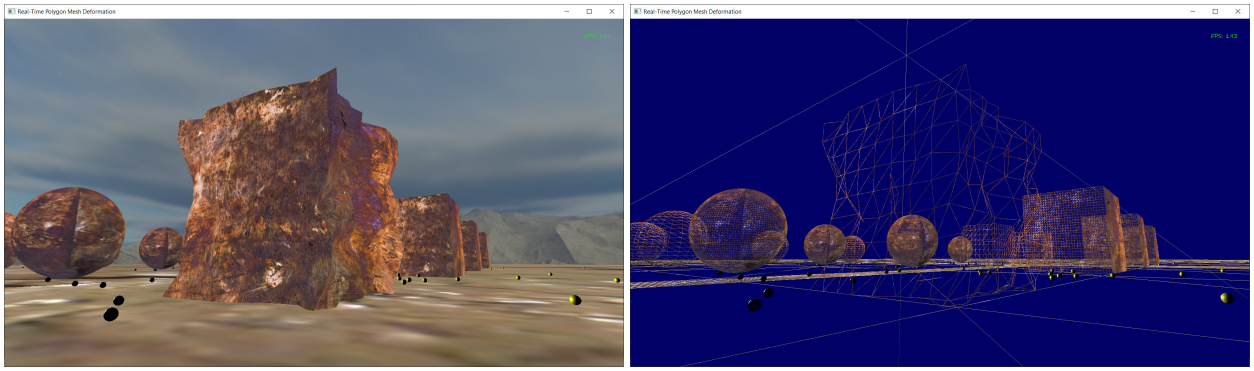
(a) Normal mode. (b) Wireframe mode.

Fig. 3: Deformed sphere having low poly mesh obtained using the uniforms array approach.



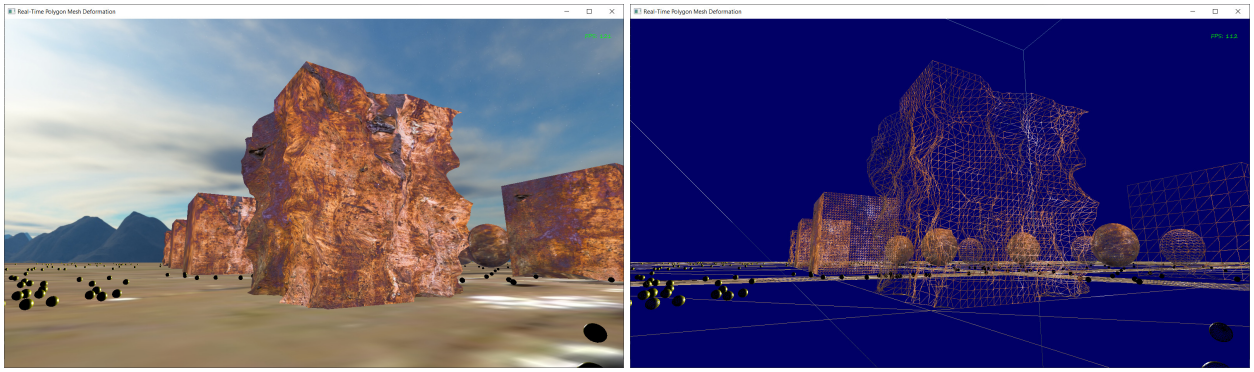
(a) Normal mode. (b) Wireframe mode.

Fig. 4: Deformed sphere having high poly mesh obtained using the uniforms array approach.



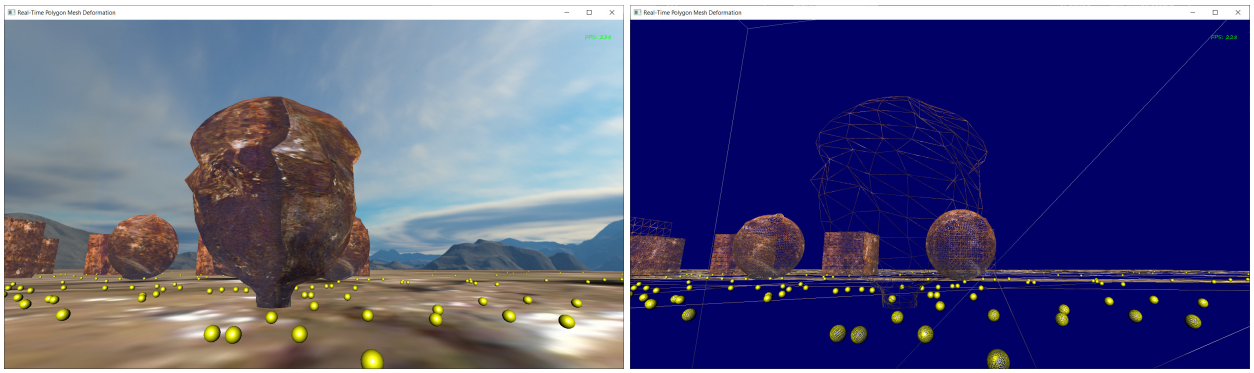
(a) Normal mode. (b) Wireframe mode.

Fig. 5: Deformed cube having low poly mesh obtained using the feedback transform approach.



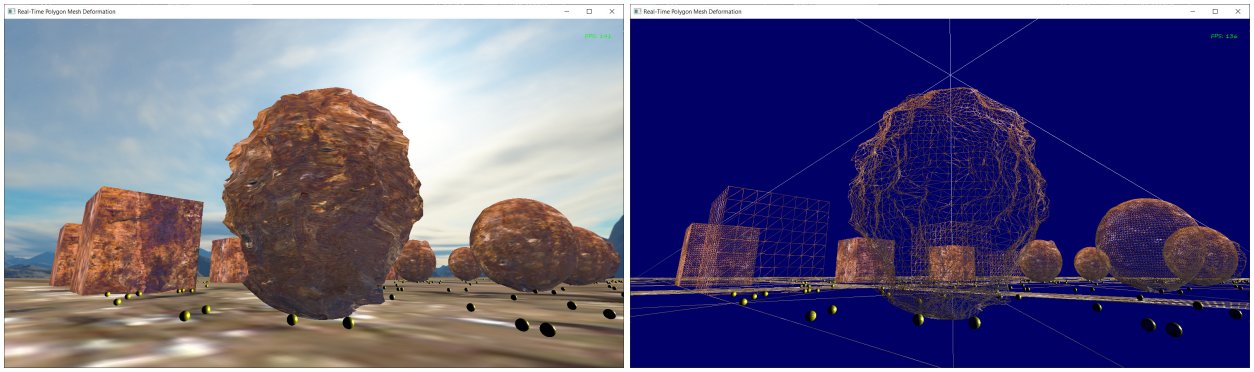
(a) Normal mode. (b) Wireframe mode.

Fig. 6: Deformed cube having high poly mesh obtained using the feedback transform approach.



(a) Normal mode. (b) Wireframe mode.

Fig. 7: Deformed sphere having low poly mesh obtained using the feedback transform approach.



(a) Normal mode. (b) Wireframe mode.

Fig. 8: Deformed sphere having high poly mesh obtained using the feedback transform approach.

REFERENCES

[1] "Learn opengl," https://learnopengl.com.
[2] "Bullet physics library," https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf.
[3] "Transform feedback," https://open.gl/feedback.
[4] "Transform feedback," https://www.khronos.org/opengl/wiki/Transform_Feedback.