Real-time Graphic Programming 2017/2018

Manuel Salvadori 886725

1. Description

The application consists in a space flight simulation. The user controls a spaceship inside an asteroid field, with the goal of dodging or destroying the asteroids as long as possible in order to survive. The spaceship can move on x and y axis by pressing WASD keys and shoot by pressing SPACE key. Every asteroid destroyed makes a point whereas being hitted leads to a flashy red screen. The total score is displayed above the hologram in the lower left corner of the screen.

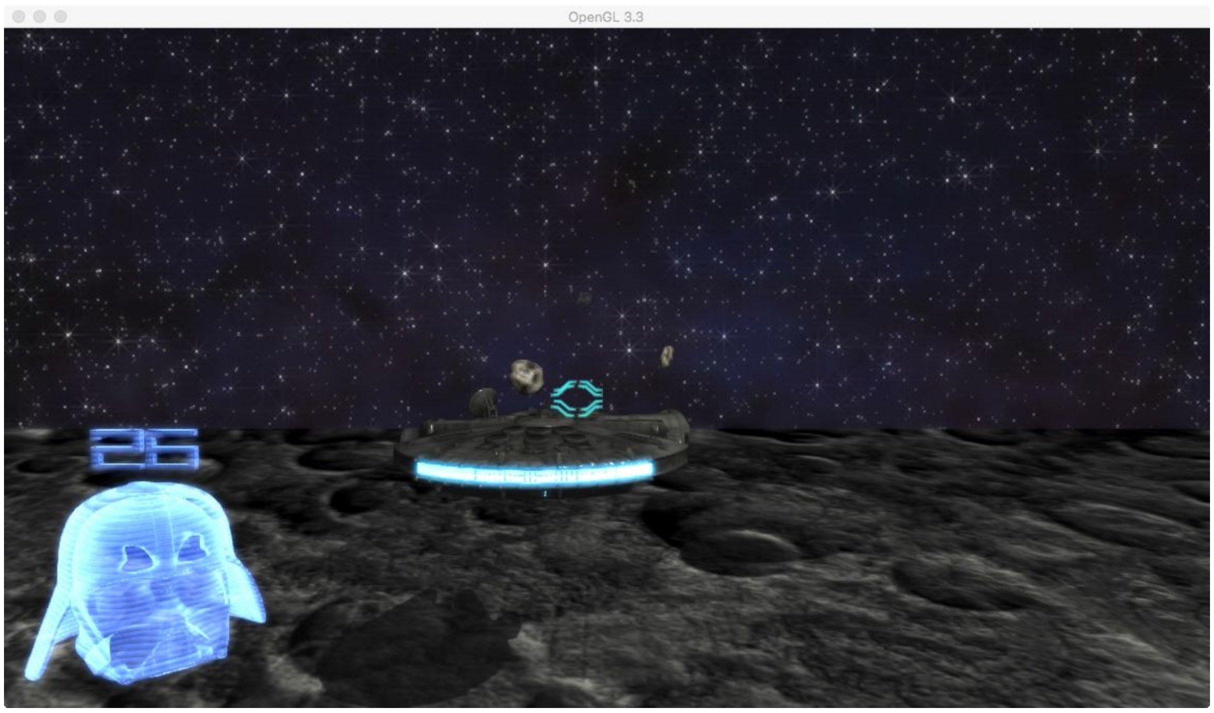

Fig. 1: Application screenshot

In general, I looked for using techniques with the lowest computational impact as possible wherever I could, since the machine I used to develop this project is not very powerful.

2. Illumination

In the scene there is only one directional light, acting like a distant sun, to better simulate an open space environment.

2.1 Blinn-Phong model

As a lighting model, I chose the Blinn-Phong reflection model. It is a modification of the Phong reflection model, where the final color is composed by three component:

- ambient color: it approximate global scattering of light in the scene
- diffuse color: it is calculated using Lambert model
- specular color: specular highlight typical of shiny materials

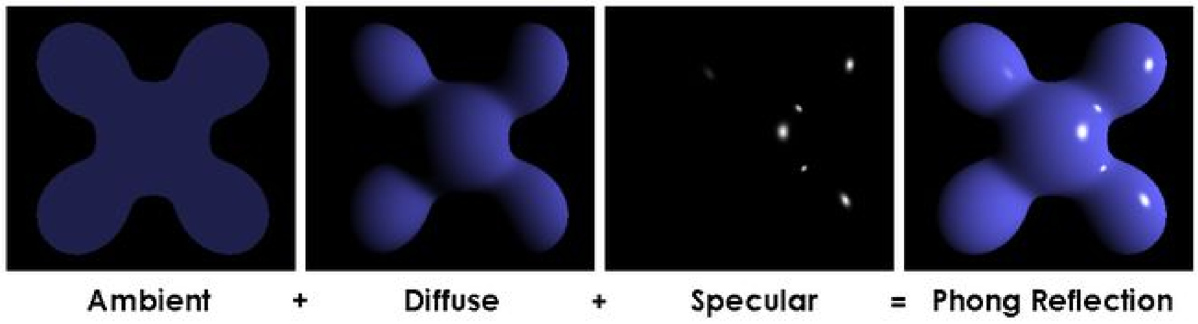

Fig. 2: Phong reflection model

The ambient color is a constant $k_a \, L_a$ whereas the diffuse color is calculated by the following formula:

$$L_d = k_d \, L_i \, cos(\theta) = k_d \, L_i \, (l \cdot n)$$

where k_d is the diffusive reflection coefficient, L_i is the incoming light from l direction and θ is the angle between the light direction and the normal direction.

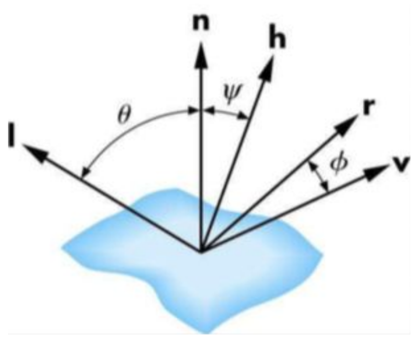

Fig. 3: Vectors involved in Blinn-Phong light calculation

The specular component is calculated based on the half vector h:

$$h = \frac{l+v}{|l+v|}$$

so the specular term in the Phong model is replaced by:

$$L_s = k_s L_i \cos(\phi)^{\alpha} = k_s L_i (l \cdot n)^{\alpha} \Rightarrow L_s = k_s L_i \cos(\psi)^{\alpha'} = k_s L_i (h \cdot n)^{\alpha'}$$

where the shiness coefficient α is modified into α' (which is determined empirically). Using the half vector speed-up the calculation, since we don't have to recalculate the r vector and we don't have to check the sign of the dot product, because ψ is always smaller than 90°.

The final color is then:

$$I = k_a L_a + L_d + L_s$$

2.2 Normal mapping

In order to improve the realism of the objects drawn, I implemented the normal mapping technique. It consists in replacing the normals of a mesh with others stored in a texture, specifically the x, y, z components of the normal vectors are saved in the r, g, b channels of the image. Usually they are generated based on a high poly version of the mesh. Another approach is to generate the normal map from the albedo texture using image processing techniques.

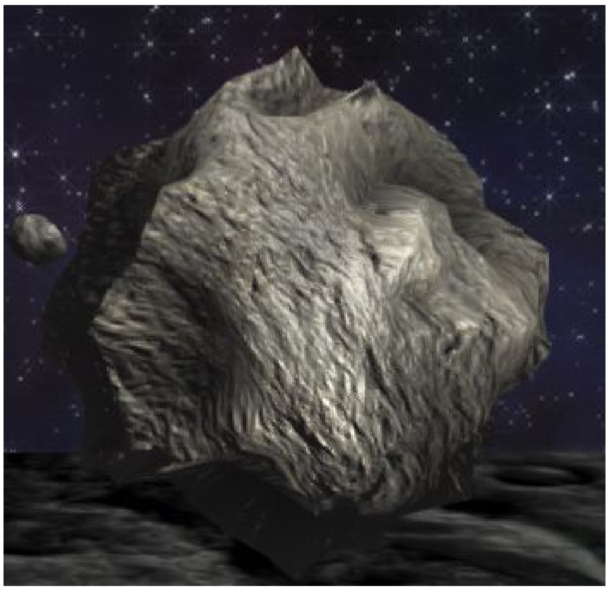

Fig. 4: An asteroid with normal mapping

The new normals are expressed in tangent space, a space that is local to the surface of a triangle: the normals are relative to the local reference frame of the individual triangles. So they must be transformed to world space in order to be used in lighting calculation.

2.3 Shadows

In order to implement shadows I used the shadow mapping technique. It consist on storing in a depth map the distance from the light source of the scene. The depth map is generated by a rendering pass from the light view. Then in the main rendering pass, the depth map is sampled to check if the point is illuminated or occluded by comparing his depth with the value stored.

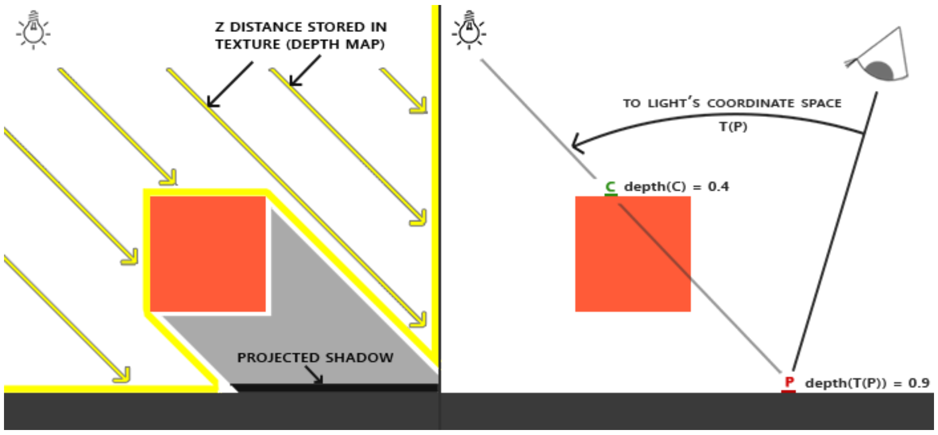
Z DISTANCE STORED IN TEXTURE (DEPTH MAP)
PROJECTED SHADOW
TO LIGHT'S COORDINATE SPACE
T(P)
C depth(C) = 0.4
P depth(T(P)) = 0.9

Fig. 5: Shadow mapping

Unfortunately, this approach generate aliasing in the shadows, dependently on the depth map resolution. A way to improve the quality of the shadow is to use the PCF (percentage closer filtering). The idea is to sample multiple time the shadow map in the neighborhood of the point considered and then average the values. This way softer shadows are produced. In particular, I used a 3x3 filter.

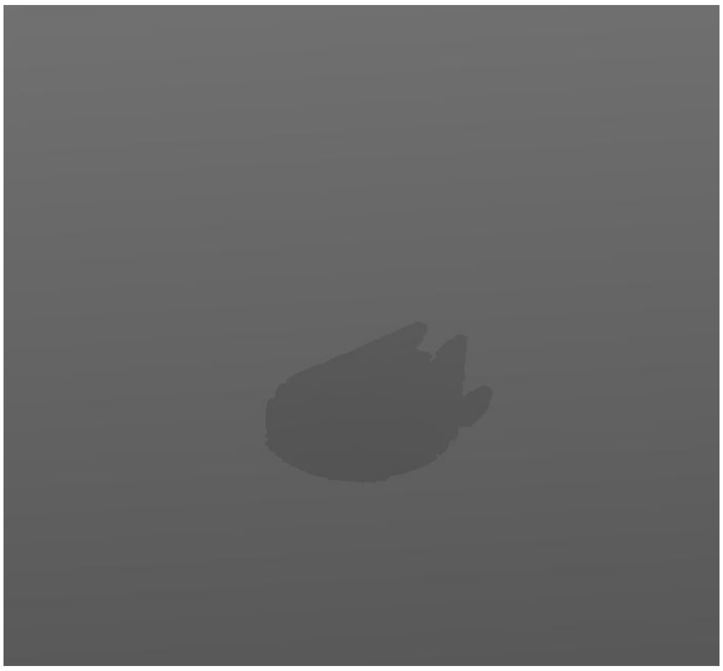

Fig. 6: Depth map

3. Post-processing

3.1 Bloom

In order to represent the glow effect of the spaceship engine and the lasers, I used a post-processing effect called bloom. Bloom can be seen by the naked eye when looking at very bright objects that are on a much darker background.

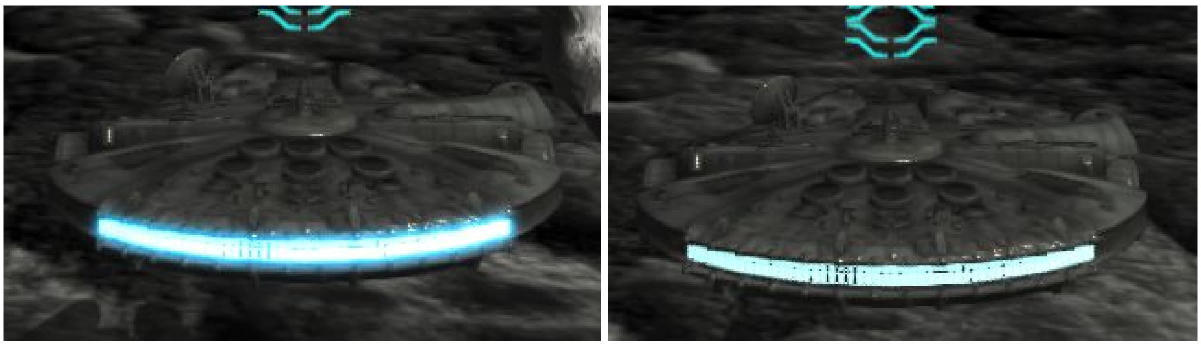

Fig. 7: Spaceship engine with and without bloom applied

The bloom effect is achieved using the Multiple Render Targets feature, which allow to render images to multiple render target textures at once. In the bloom case, in a single render pass, we can obtain two images, one with only the brightest fragments, the other with the regular color. The selection of the brightest fragments is done by calculating their luminance, transforming the color to grayscale first (by taking the dot product of the color vector with a constant vector). If it exceeds a certain threshold, we output the color to the second colorbuffer that holds all bright regions.

$$float\ brightness\ =\ dot(FragColor.rgb,\ vec3(0.2126,\ 0.7152,\ 0.0722));$$

Another approach is to select only the fragments that have an actual emissive component. We can even change the color of the bright fragments, as I did for the lasers. The base color of the lasers is white, whereas its bloom color is red. This way we can simulate the incandescence of a material (the more hot is a material, the more the light emitted is white at the surface, fading away to red).

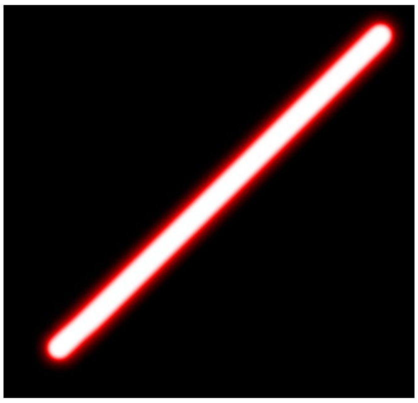

Fig 8: Laser with bloom effect

Next, using a second render pass, the render texture storing the brightest region of the scene is blurred with a gaussian blur technique. To obtain better result, we can do this step multiple times. In this project I repeat the blur render pass three times using a two-pass gaussian blur (so in total are six render pass) with a 7x7 pixels mask. The two-pass gaussian blur consist in blurring separately along the horizontal axis and the vertical axis. This technique is considerably faster with respect of the regular gaussian blur, with almost the same quality.

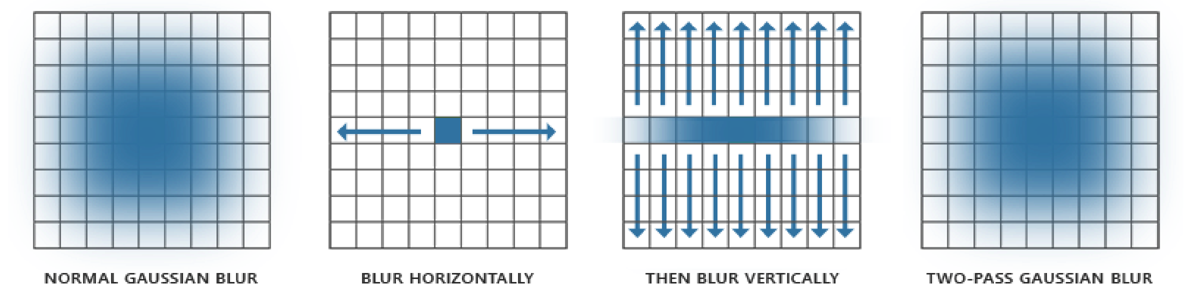

Fig 9: Two-pass gaussian blur

Finally, a last render pass is needed in order to merge the regular image with the blurred one, by simply adding their color pixel by pixel. At this stage I also perform the post-processing anti-aliasing (only on the regular color render texture, since the bloom one is blurred and hence not affected by aliasing).

3.2 FXAA

As an anti aliasing algorithm I choose the FXAA (fast approximate anti aliasing), first designed by nVIDIA. This is a post-processing technique, well known for its very low computational impact. Among those, subpixel morphological anti-aliasing (SMAA) is one of the most state-of-the-art ones, but is quite complex to implement. A much simpler but still extremely efficient algorithm was described by Timothy Lottes from nVIDIA in 2009 and has quickly made its way into many games: FXAA.[1]
In summary, this algorithm perform an edge detection on the image and apply only to them a low-pass filter. More in detail, FXAA convert the RGB color into a scalar estimate of luminance and then checks local contrast to avoid processing non-edges. Pixels passing the local contrast test are then classified as horizontal or vertical. Given edge orientation, the highest contrast pixel pair 90 degrees to the edge is selected. The algorithm searches for end-of-edge in both the negative and positive directions along the edge, checking for a significant change in average luminance of the high contrast pixel pair along the edge. Given the ends of the edge, pixel position on the edge is transformed into to a sub-pixel shift 90 degrees perpendicular to the edge to reduce the aliasing. The input texture is re-sampled given this sub-pixel offset. Finally a lowpass filter is blended in depending on the amount of detected sub-pixel aliasing.
The efficiency, compared to MSAA or other super/multi sample AA, is very high but unfortunately some texture details are lost, especially when we have sharp details (because of the edge detection and blur). The overall quality is really good anyways.

[1] http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf

4. Asteroids

4.1 Procedural generation

The asteroids are procedurally generated via shader using a vertex displacement technique on a spherical mesh. In the vertex shader each vertex of the sphere mesh is perturbed by a perlin noise function along its normal. I used the perlin noise implementation by Ashima Arts (github.com/ashima/webgl-noise).

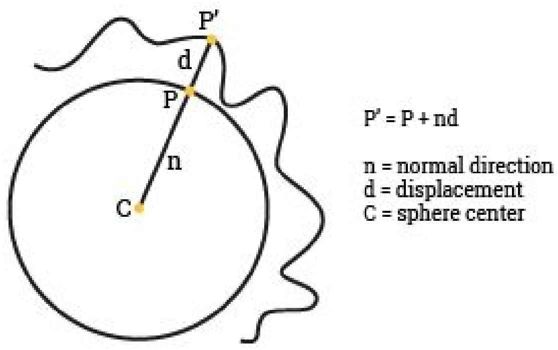

Fig. 8: Vertex displacement

The displacement amount is calculated by sum two terms, one using a low frequency noise that shape the asteroid in a general way, the other using an high frequency noise for adding details. Both take account of two random terms p_{1-4}:

$$low = p1 * pnoise(p3 * aPos, vec3(100.0))$$
$$high = p2 * pnoise(p4 * aPos, vec3(10.0))$$
$$newPos = Pos + (normal * abs(low + high))$$

The whole process is applied in local space, before any transformation. The noise amount is then passed to the fragment shader, where is used to simulate a fake ambient occlusion, by darken the concave part of the generated asteroid. To further improve the diversity between the asteroids a random not uniform scale and a random grey color shift (desaturation) is also applied.

4.2 Object pooling

To minimize the computational impact of the asteroid field, I implemented the object pool creational pattern. Instead of continually allocating and destroying the asteroid objects on demand, I just preallocate a fixed number of objects (specifically only four). Every time an asteroid is out of the camera frustum or destroyed by the player, it is recycled recomputing the random terms of the procedural generation. This way we can have an infinite amount of different asteroids with only the allocation cost of four objects. The only downside is that we can only have a number of objects, present at the same time, equal to the object pool size. I then applied the same pattern also to the spaceship lasers with a pool size of ten objects.

5. Hologram

In the lower left corner of the screen I added a scorekeeper shaped as an hologram. To obtain the hologram look I used both vertex and fragment shaders. The main color is given by the Lambert model with transparency, later modified by the following effects:

5.1 Glitch

To simulate the glitch effect typical of the holograms, caused by a disturbed transmission, I applied a random vertex displacement along the x axis in world frame. The amount is given by a step function driven by a sine over time, multiplied by a sign function:

$$amount = sign(sin(time)) * step(0.5, sin(time * 2.0 + vertexLocalPos.y))$$

5.2 Flickering

A disturbed transmission, other than glitch the shape of the hologram, should add a noisy color. To do that, I used once again a perlin noise function, based on time and world coordinates, that alter the alpha channel of the fragment color:

$$min = 1 - step(0.2, sin(time * 2.0)) * step(0.01, sin(time * 0.5));$$

$$flicker = clamp(pnoise(vertexWorldPos * noise(time * flickerSpeed), vertexWorldPos/100), min, 1.0)$$

5.3 Horizontal bars

I used a smoothstep function in order to give a striped look to the hologram, based on the y position in world coordinates:

$$bars = smoothstep(fract(vertexWorldPos.y * barDistance), 0.2, 0.5)$$

5.4 Scan effect

Last there is a scan effect that lower the brightness of the final color based on the y position in world coordinates and the time:

$$scan = -fract(vertexWorldPos.y + time * scanSpeed)$$

Finally I added a little bit of bloom effect to give to the hologram a glowish appearance.

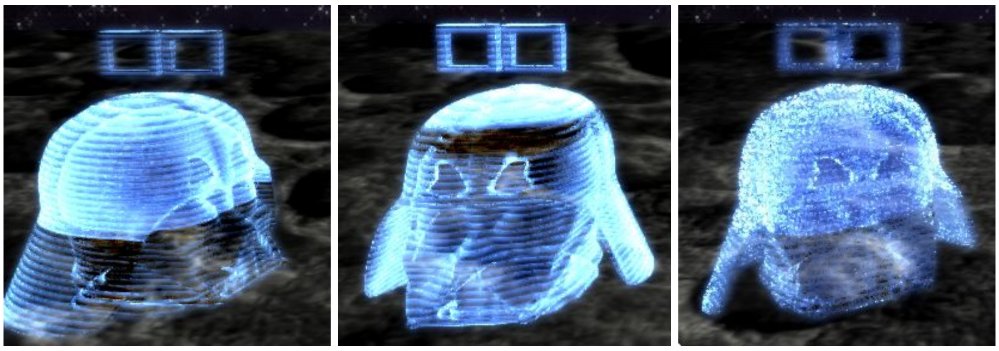

Fig. 9: Hologram

6. Collision detection

Collision detection can be really computational expensive, so I had to choose a simple approach. The simpler one is to use bounding spheres and spheres intersection tests; this approach lends itself well since the asteroids have a spheroidal shape. To calculate the bounding sphere radius I considered the radius of the base sphere mesh, and I scaled it with the mean of the random scale values along the three axis, plus an average displacement value given by the perlin noise. Those values are generated in the procedural generation process of the asteroids.

$$radius = baseRadius * (scale.x + scale.y + scale.z) / 3 + averageDisplacement$$

This way we have a better approximation of the objects' radius. I applied the same process to the spaceship, whereas the lasers are treated as point shaped objects, so spheres with degenerate radius. An intersection test then is simply given by:

$$hitted = spheresDistance < (radius_1 + radius_2)$$

which is extremely cheap on computational demand, at the cost of have lesser collision detection accuracy when the asteroids shapes are particularly oblong.
The result of a collision between an asteroid and the ship is the triggering of a flashy red screen, suggesting the danger. The result of a collision between a laser shooted and an asteroid instead is his own destruction. The explosion is pictured by an animated texture.

7. Performance

This project has been developed and tested on a MacBook Pro 13 late 2011 laptop with the following specifications:

CPU	Intel Core i5 2.4 GHz
Memory	4 GB 1333 MHz DDR3
Graphics	Intel HD Graphics 3000 512 MB integrated card

This machine is very not suitable for graphic calculations, its integrated graphic card have only 12 shader cores instead of the thousands present in most of the recent GPUs. As a matter of fact, although all of the optimization I made, the frame rate is pretty low:

Screen resolution	Framerate
806x453	20/24 fps
1152x648	17/19 fps