

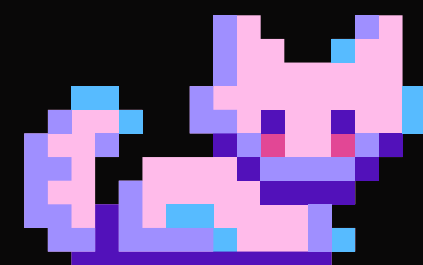
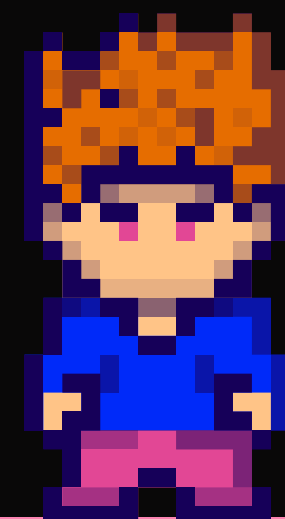
MENU

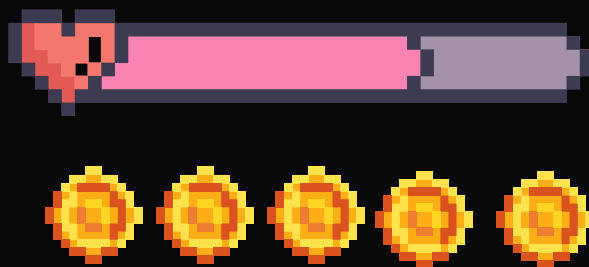
START



KOSS PROJECT

OpenGL shaders



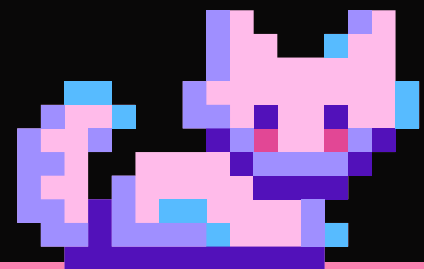
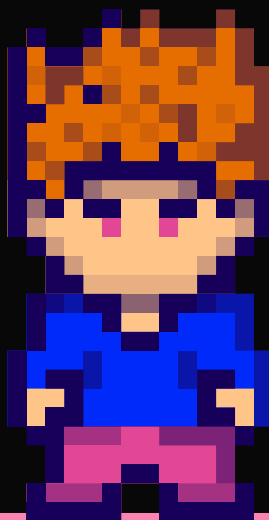


MENU

START



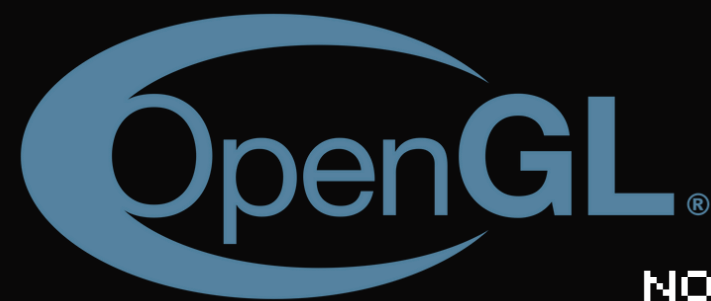
INTRODUCTION



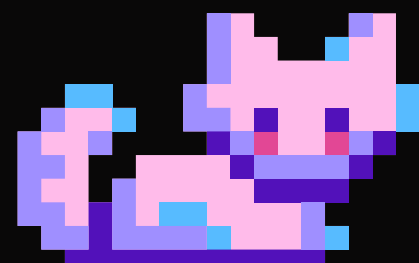
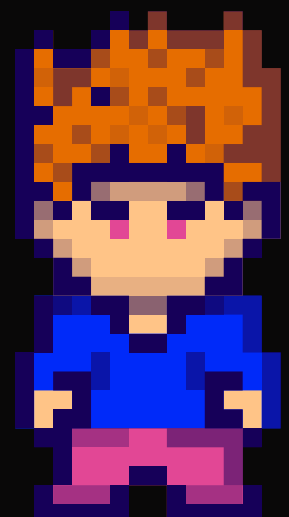


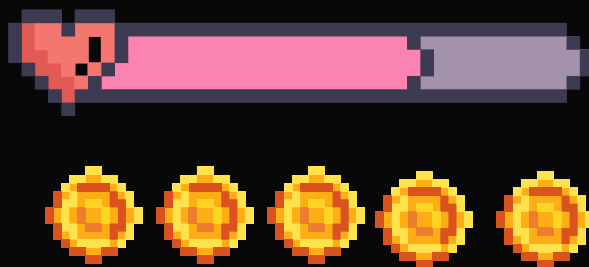
WHAT IS OPENGGL?

OPENGL (OPEN GRAPHICS LIBRARY) IS A
CROSS-PLATFORM, HARDWARE-ACCELERATED
API (APPLICATION PROGRAMMING INTERFACE)
USED FOR RENDERING 2D AND 3D VECTOR
GRAPHICS.



NOW LETS START WITH SOME
BASIC ELEMENTS >>>



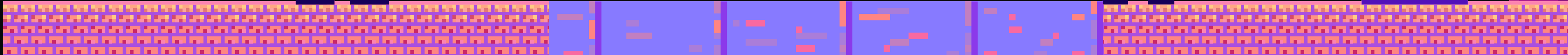
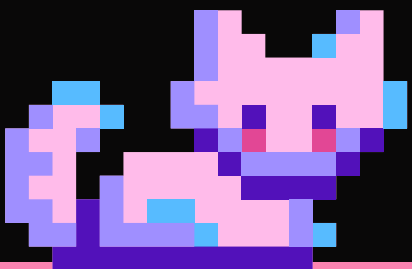
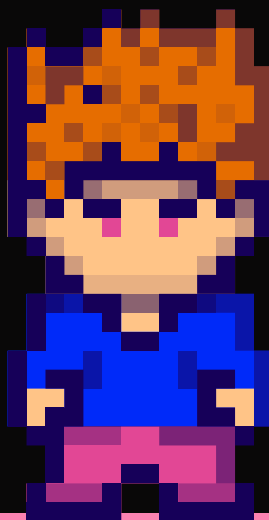
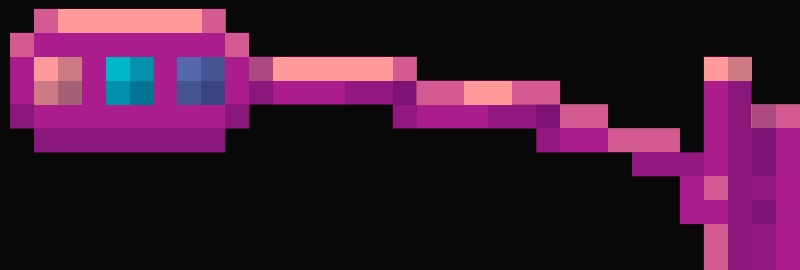


MENU

START



BASICS

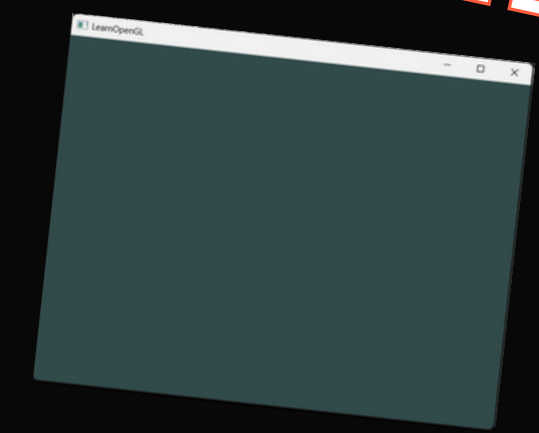




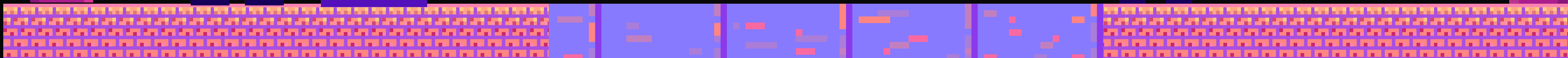
HELLO WINDOW !

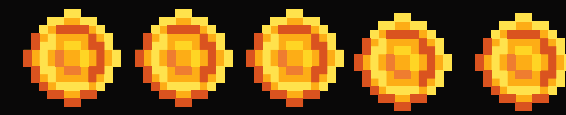
An artist might not always need a stage to
perform but we need a window to
showcase our work.
Starting with pur Window.

WHAT I MADE:



hehe !



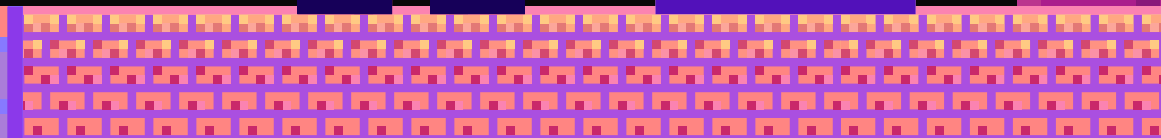
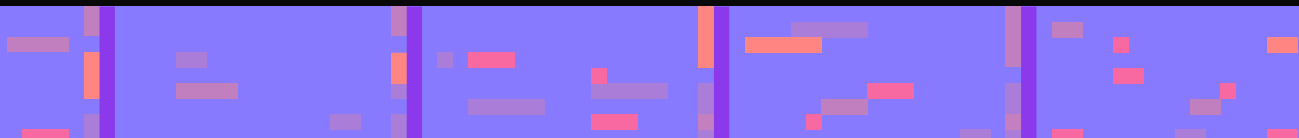
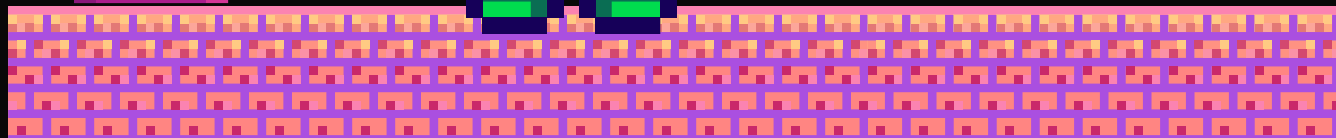
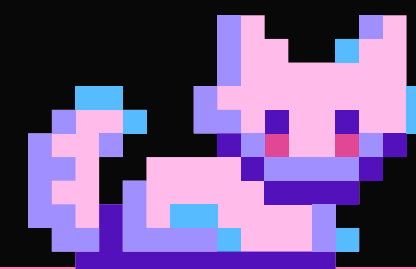
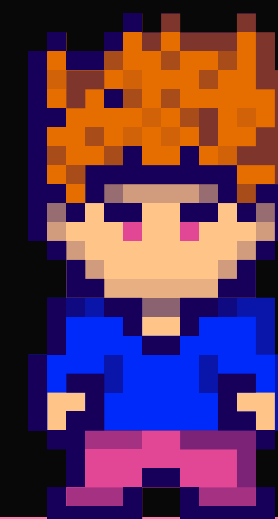
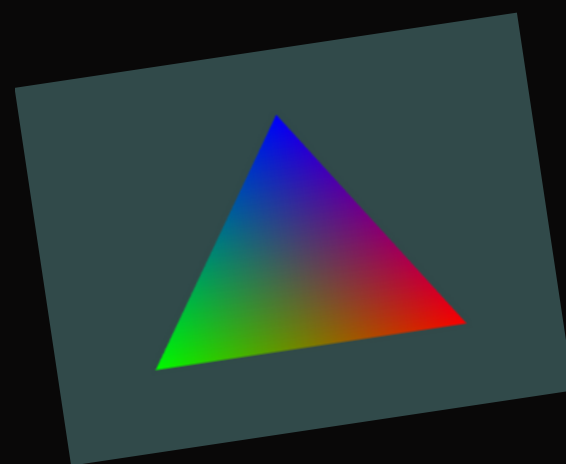
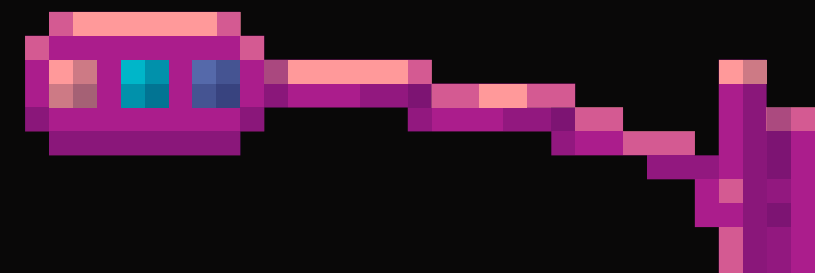


AHEAD: 2D SHAPES



starting with a little bit of graphics :

some big terms : GRAPHICS PIPELINE & SHADERS.





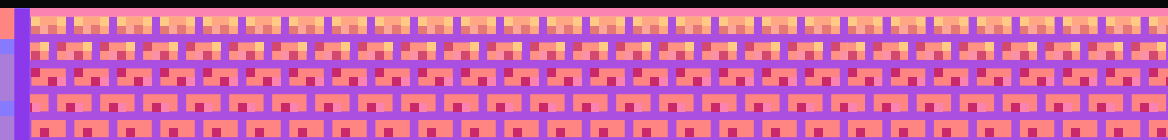
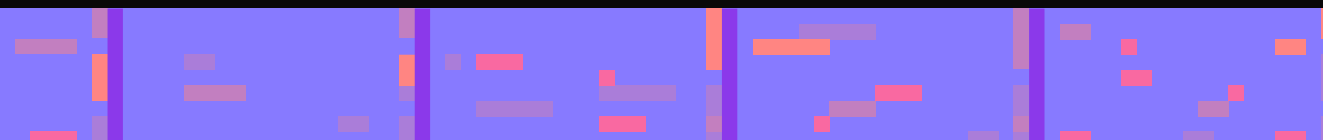
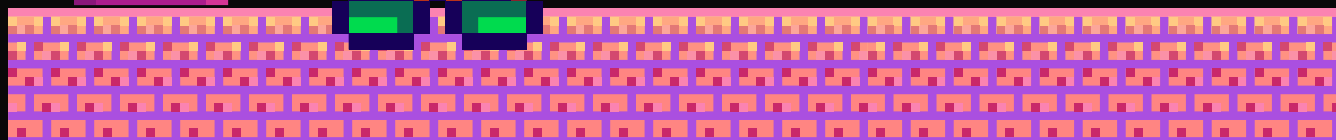
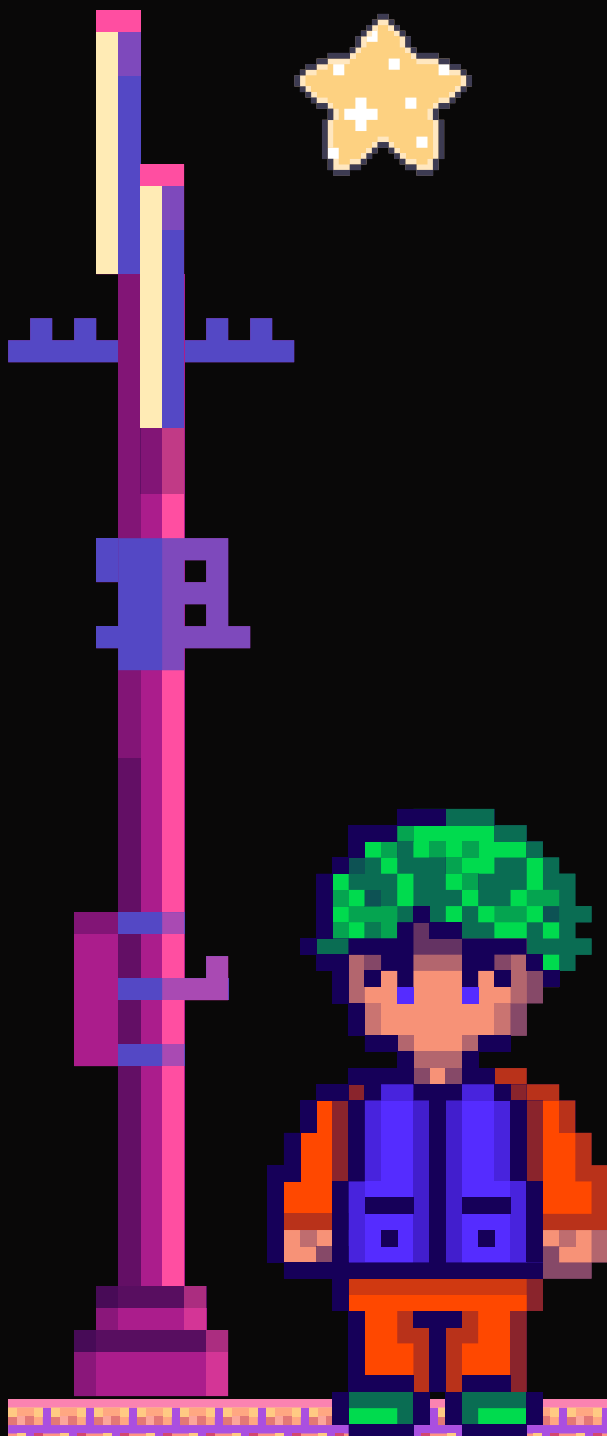
GRAPHICS PIPELINE:

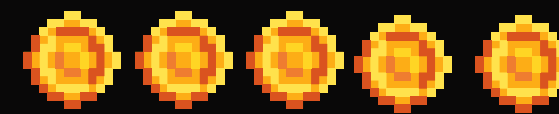


Converting a 3d world space coordinate in a 3D space to a 2D coordinate (pixels) on our screens is quite of a job to do and involves too much maths and brain. Here is where Graphics Pipeline come in play.

Formal Defination:

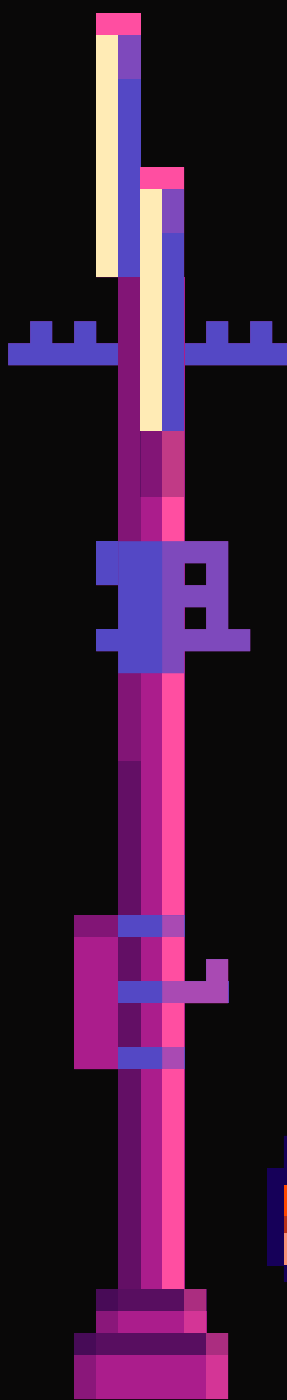
"The Graphics Pipeline is the sequence of steps used by OpenGL (and the GPU) to transform 3D models into a 2D image on our screen."

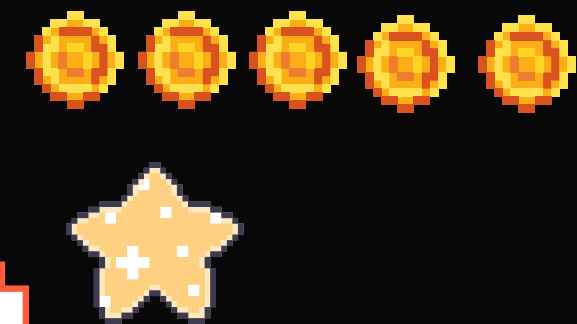




WHAT ARE THE STEPS ???

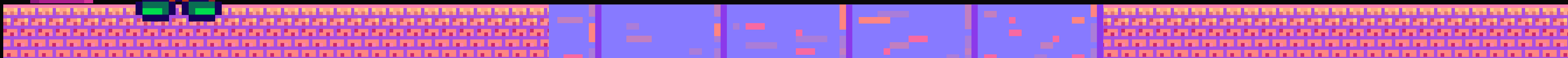
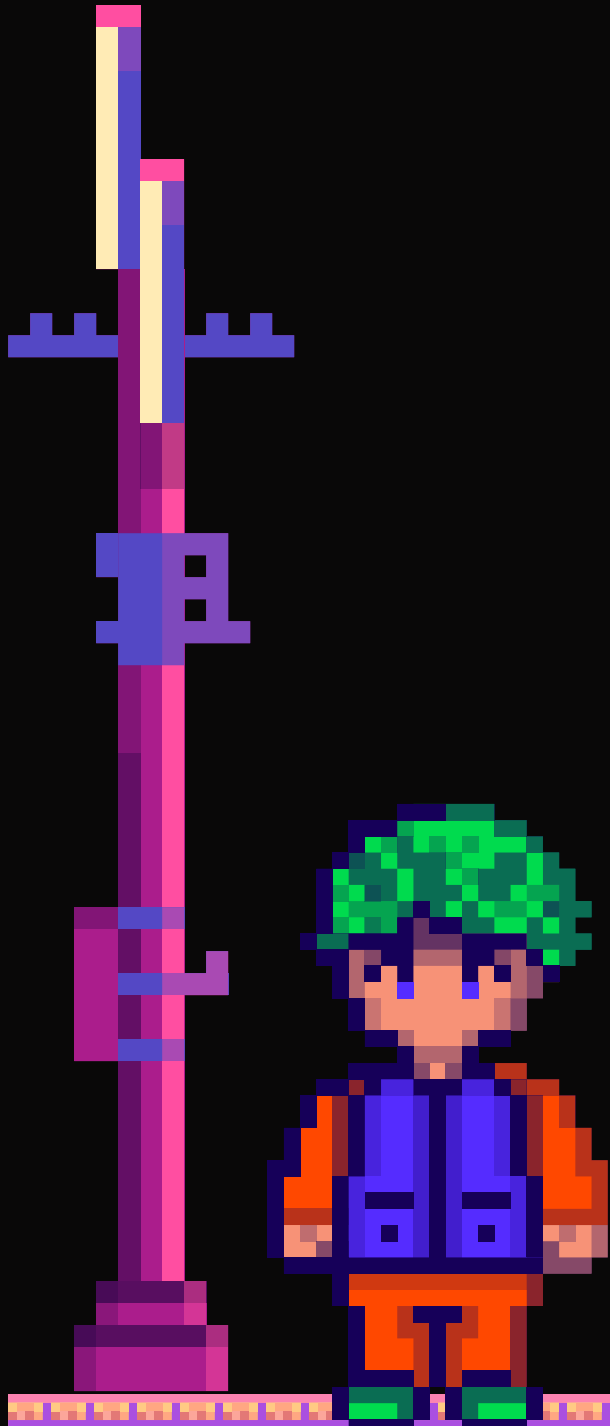
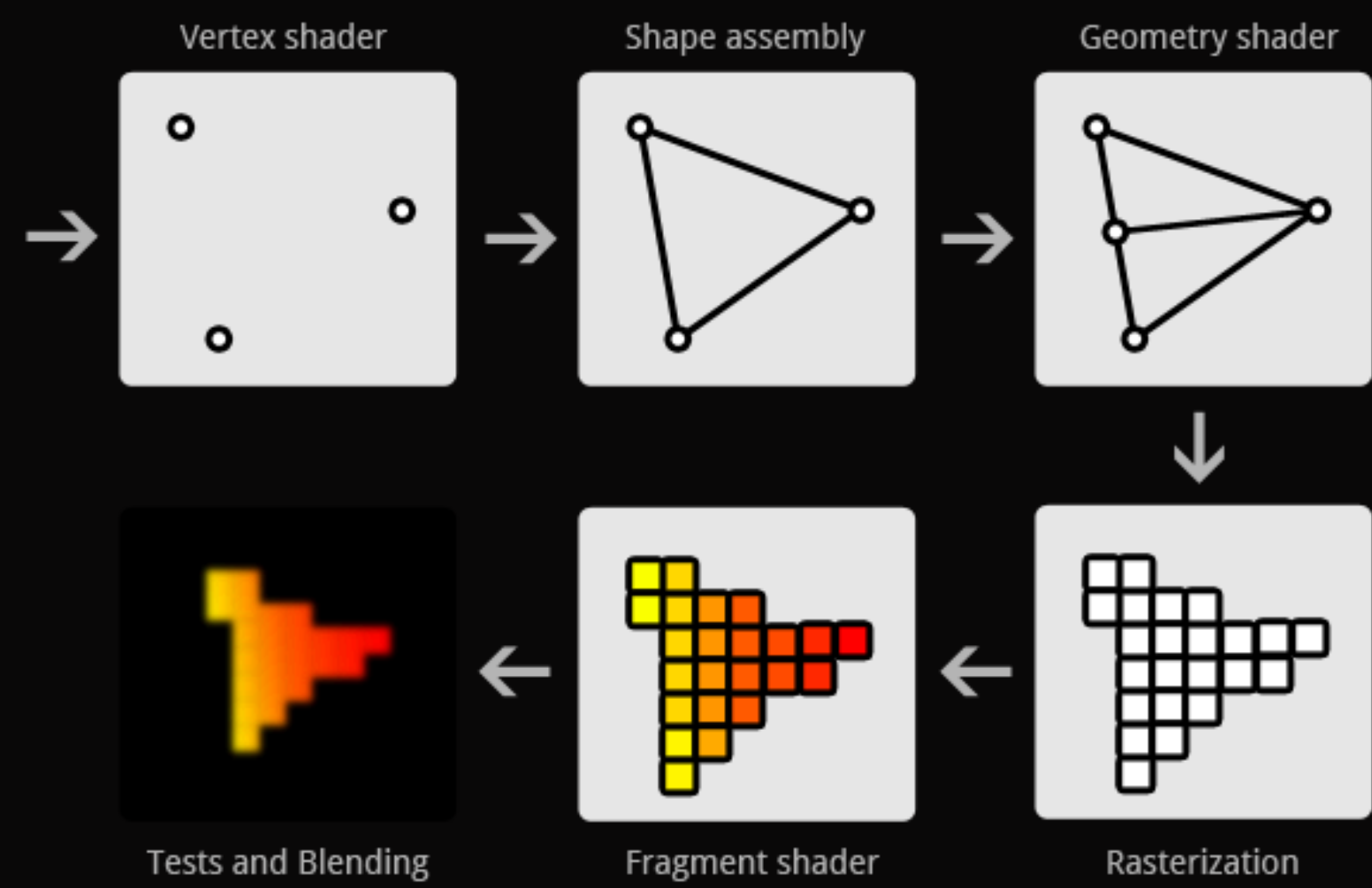
1. taking the points (called vertex) of the 3D object.
2. assigning it different locations on the screen.
3. connecting the dots (points)...
4. turning the shapes into tiny squares called fragments (like pixels).
5. giving each pixel some colour
6. can add effects like lighting, shadows and textures and stuff
7. finally combining the pixels and done,

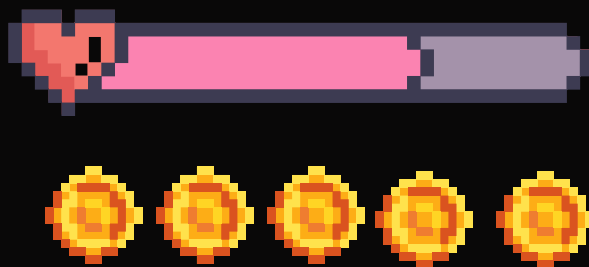




LET'S DIVE DEEPER

but before doing that we need to know about shaders >>>



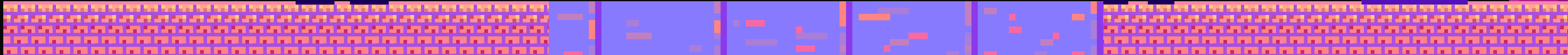
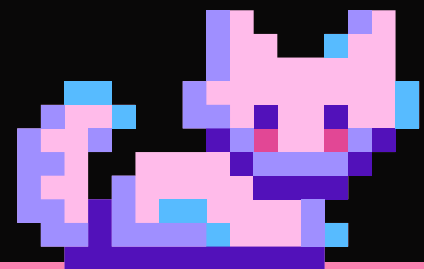
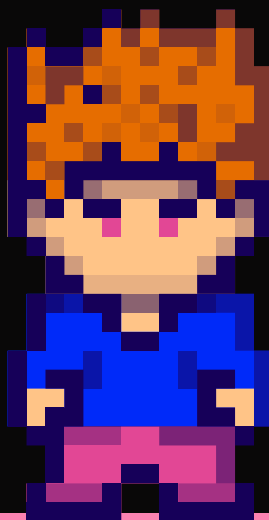
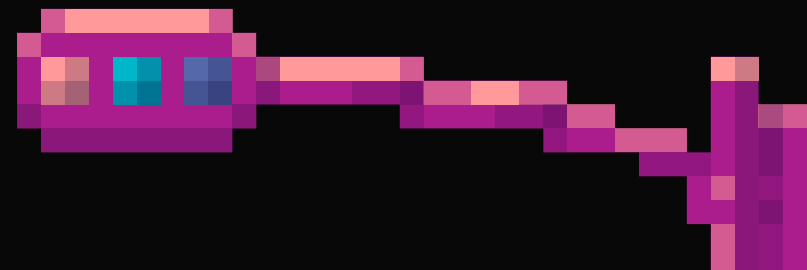


MENU

START



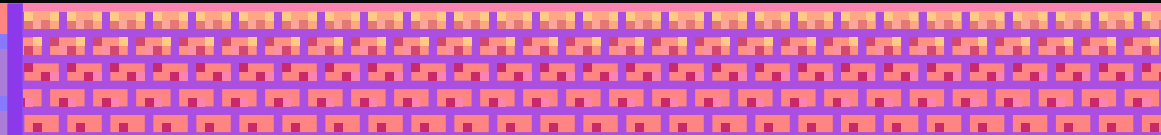
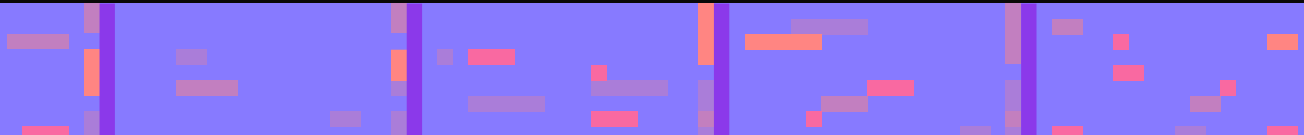
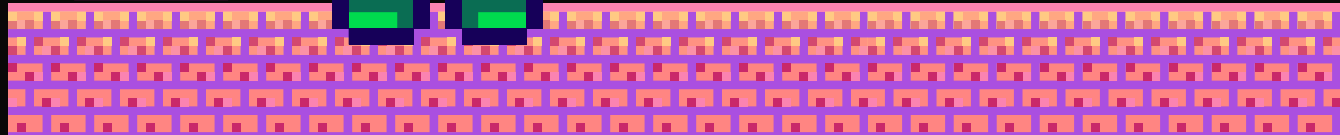
SHADERS

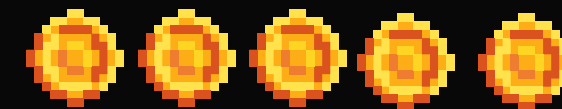




Shaders are small programs that run on the GPU.
It runs for each specific section of the graphics pipeline.
They control how our 3D objects look — their shape, color, lighting,
and more.

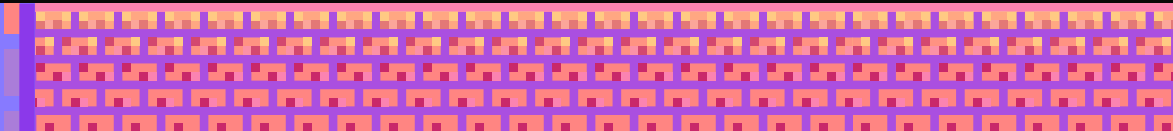
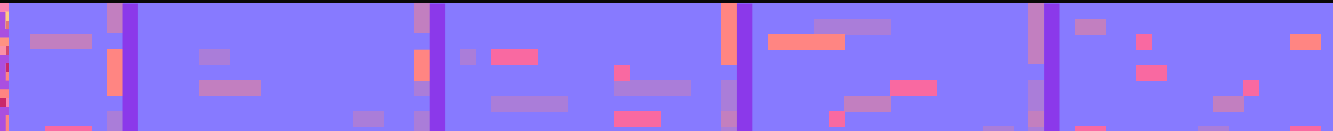
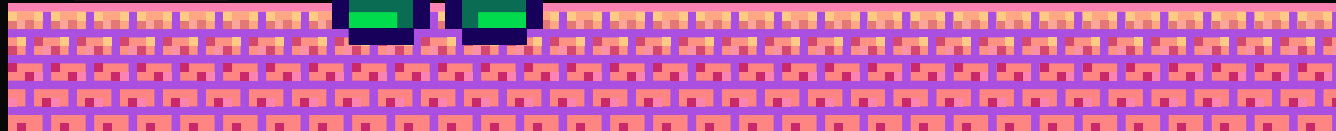
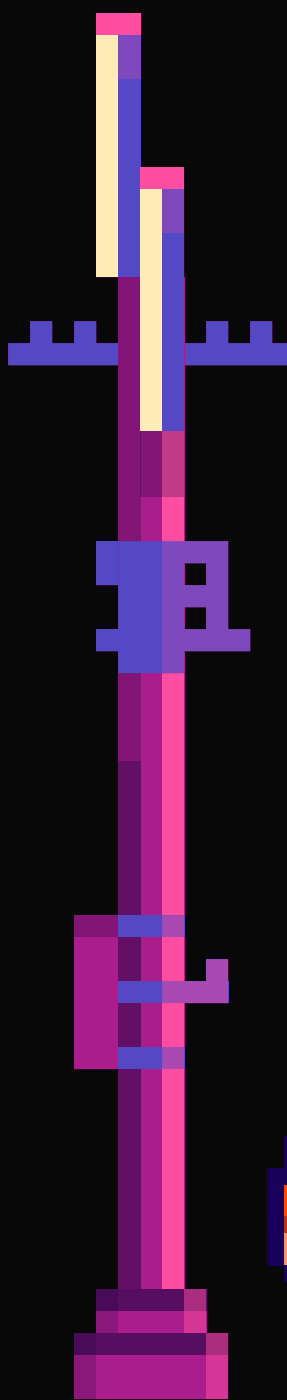
It runs on the graphic card's processing cores which are all run at once
parallely.
These shaders are written in GLSL.

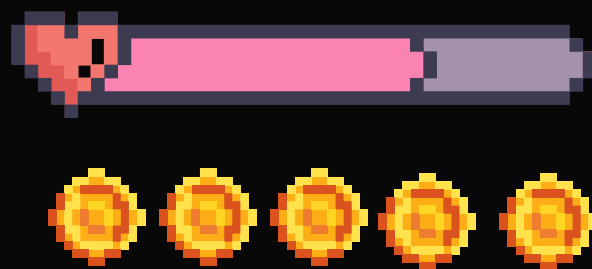




Q: WHY USE SHADERS ?

1. Make objects look real with lighting and shadows.
2. Add effects like glow, blur, reflections, or color changes.
3. Make graphics fast and flexible because they directly run on the GPU (exactly).



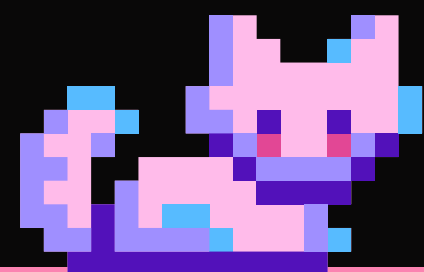
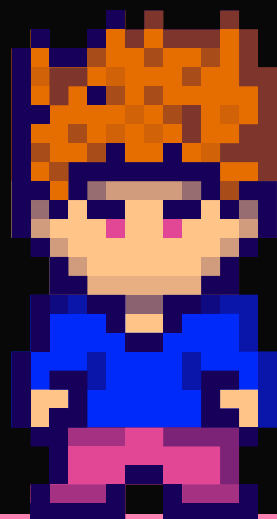
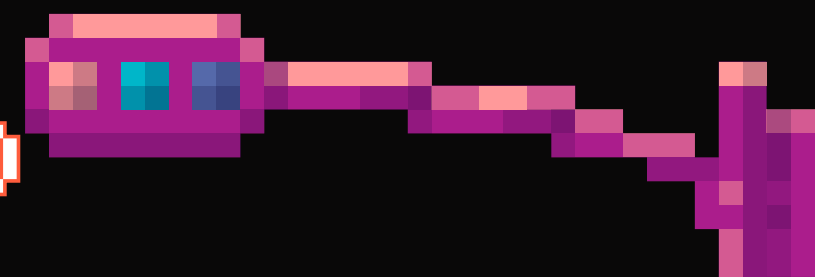


MENU

START



BUT HOW TO TELL OPENGL WHERE TO
KEEP MY OBJECT??





VERTEX SHADERS

Tells OpenGL to where each vertex should appear on the screen



Takes in: Position, color, texture, etc. of a vertex.



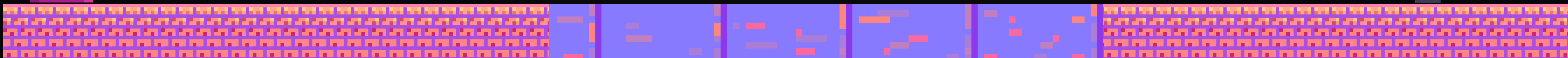
Performs math: Moves, rotates, or scales the vertex using transformation matrices.

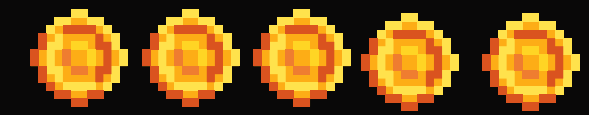


Outputs: The new position of that vertex on the screen.



Fragment shader does the work ahead ...





COMMON INPUTS



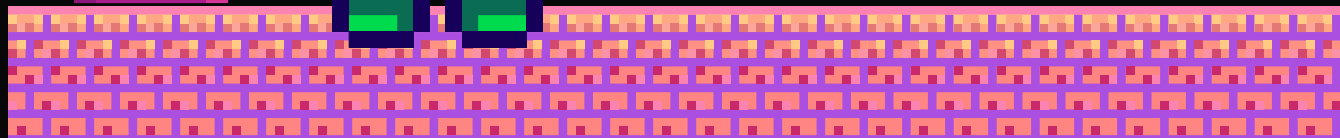
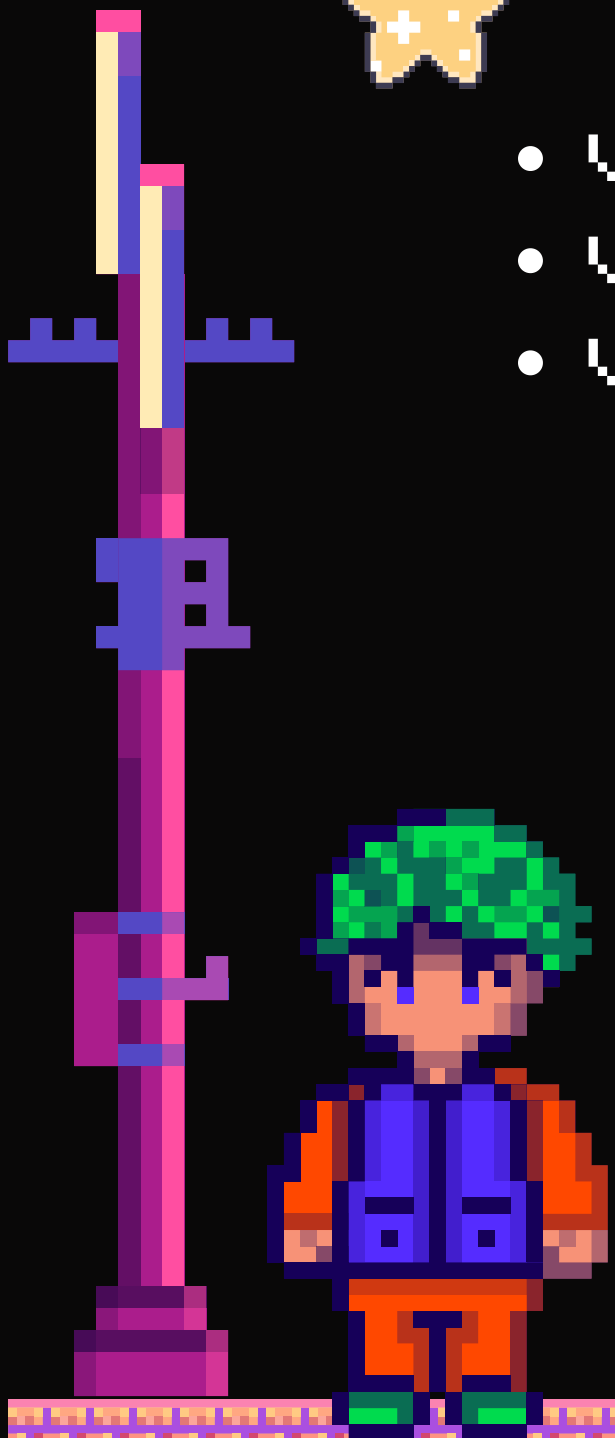
- vec3 position – The 3D location of the vertex (x, y, z).
- vec3 color – The color at that vertex (r, g, b).
- vec2 texCoord – The texture coordinate (u, v) for applying images.

```
#version 330 core
```

```
layout (location = 0) in vec3 aPos;    // input: vertex position
```

```
layout (location = 1) in vec3 aColor;  // input: vertex color
```

```
layout (location = 2) in vec2 aTexCoord; // input: texture coordinate
```



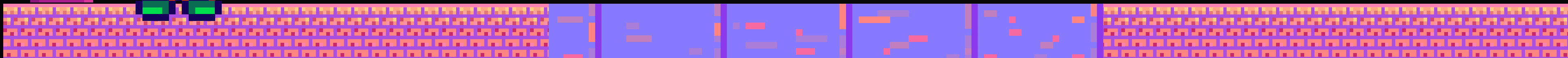


TRANSFORMATIONS



Transformations are used to move, rotate, and scale your 3D objects so they appear correctly in the scene.

The vertex shader takes the position of each vertex and applies math operations (matrices) to transform it through stages:

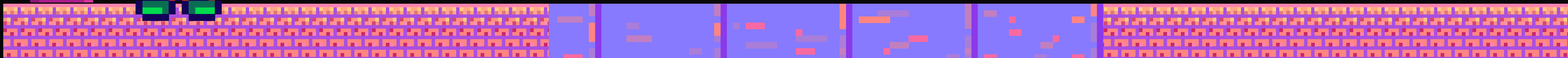
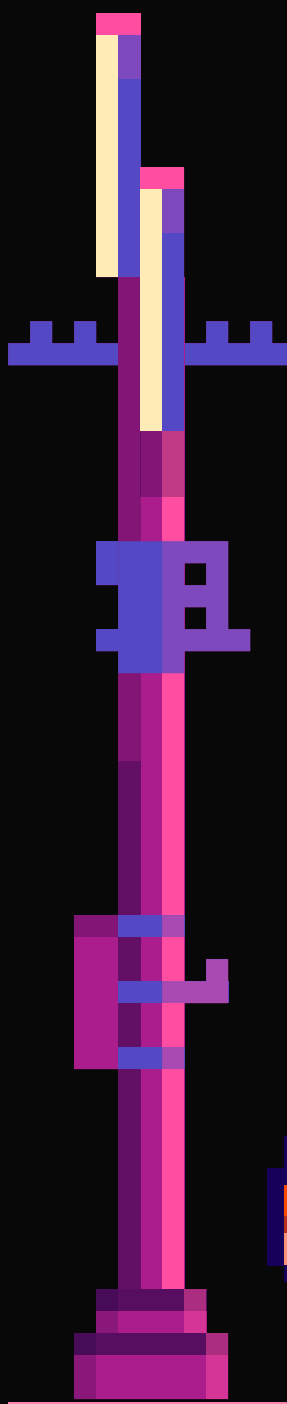




MODEL

Moves the object from local space (its own position) into the world.
It combines translation, rotation and scaling.

```
vec4 worldPos = modelMatrix * vec4(aPos, 1.0);
```

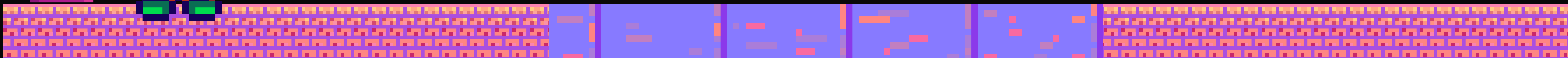
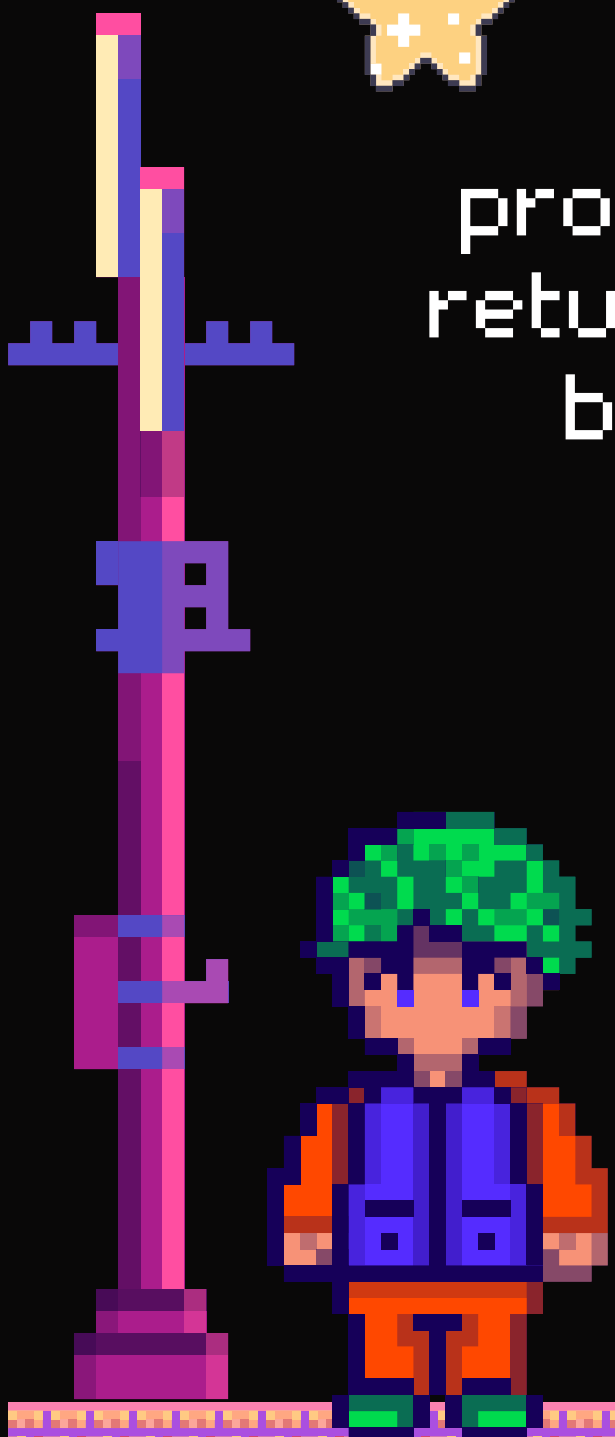




I. TRANSLATION

process of adding Translation vector on top of the original vector to return a new vector with a different position, thus moving the vector based on a translation vector. Where T is the translation vector.

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

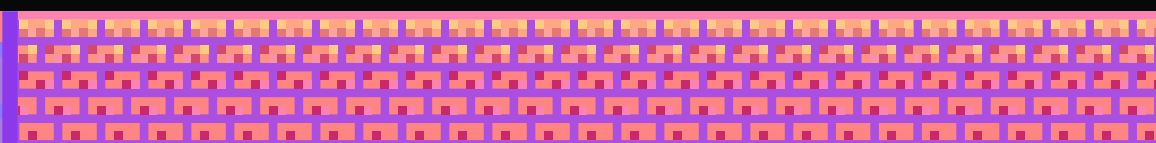
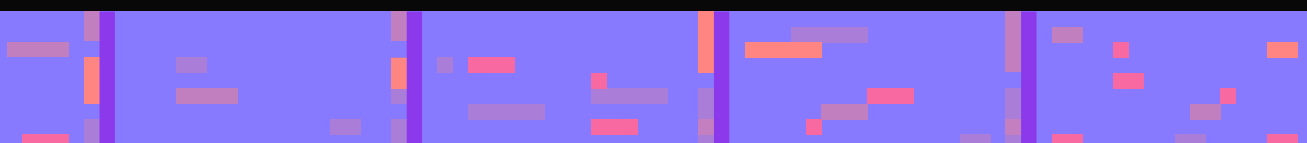
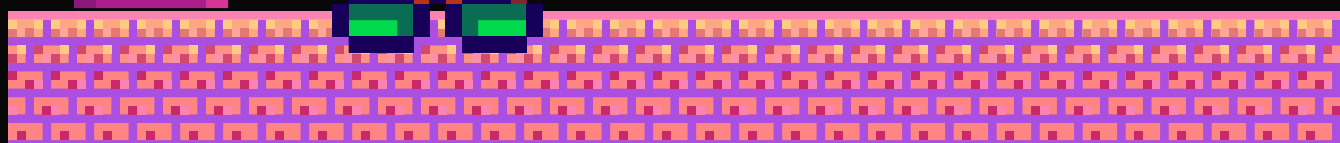
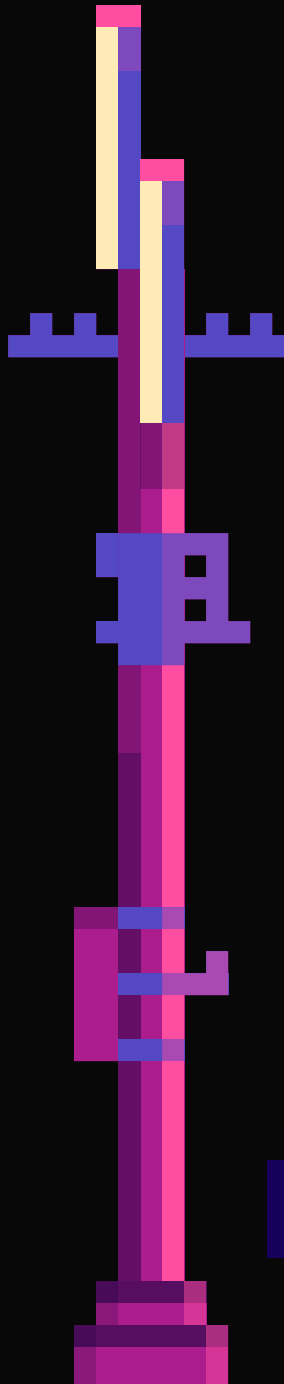




II. ROTATION

Rotation means spinning an object around an axis (X, Y, or Z).

- X-axis rotation tilts the object up/down
- Y-axis rotation turns the object left/right
- Z-axis rotation spins the object like a clock face





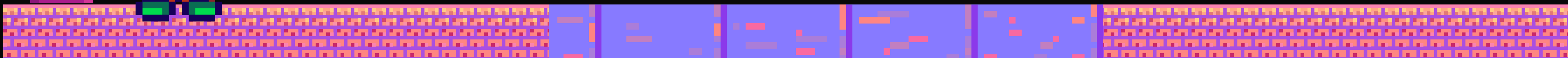
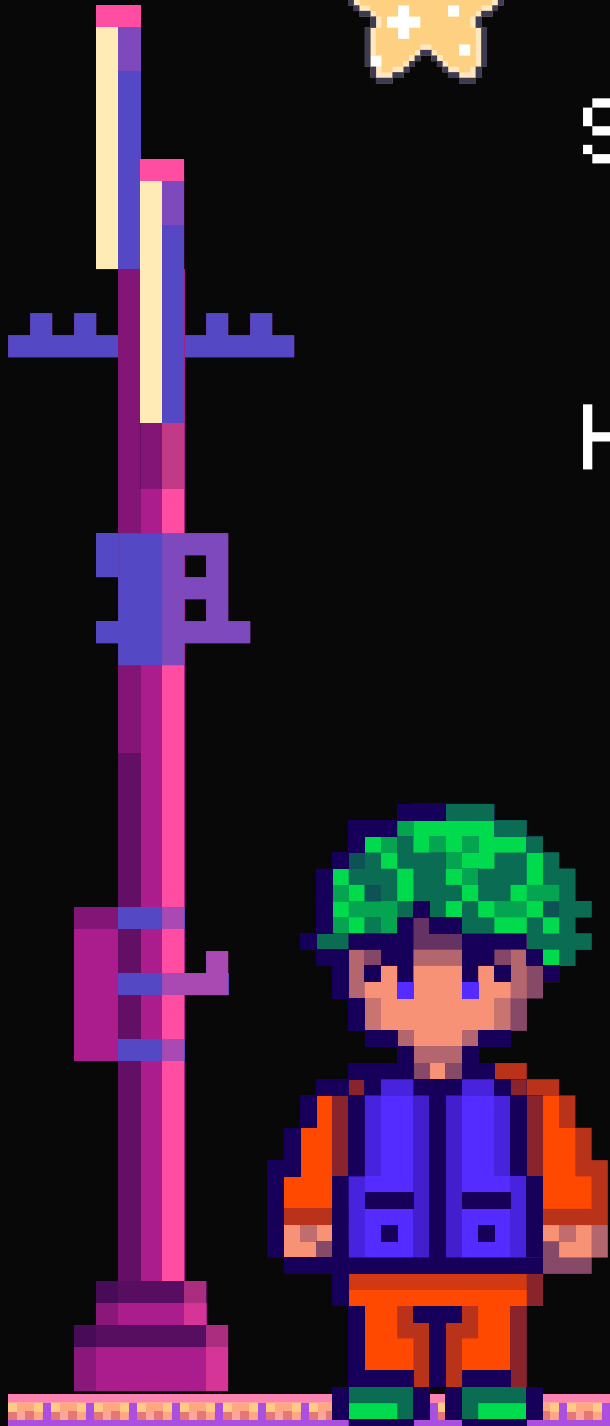
III. SCALING

Scaling means resizing an object — making it bigger or smaller.

- Uniform scaling: all directions grow/shrink equally
- Non-uniform: different scales on X, Y, Z.

Here, S is the scaling vector.

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

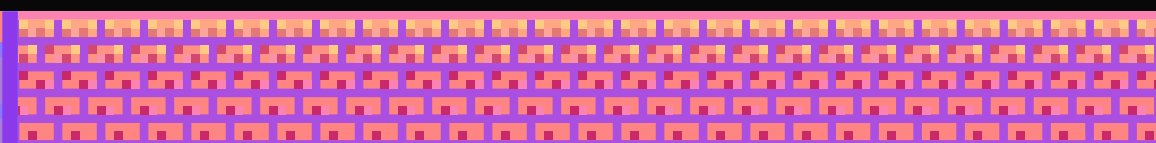
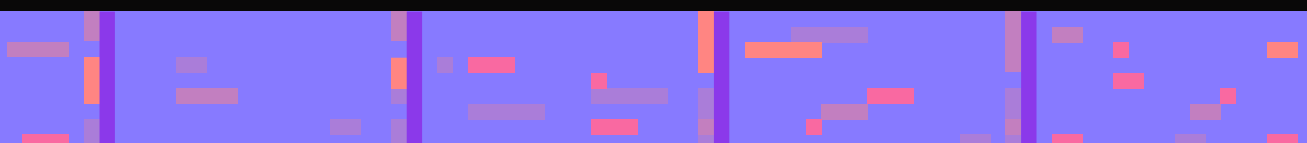
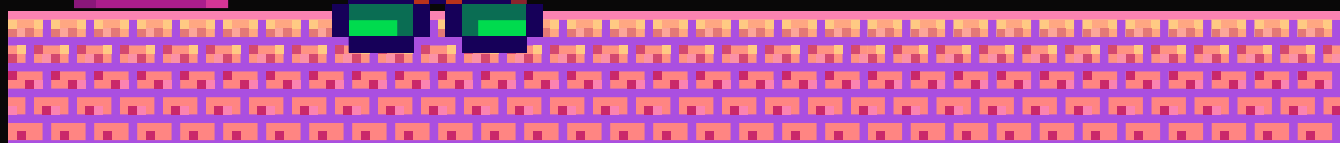
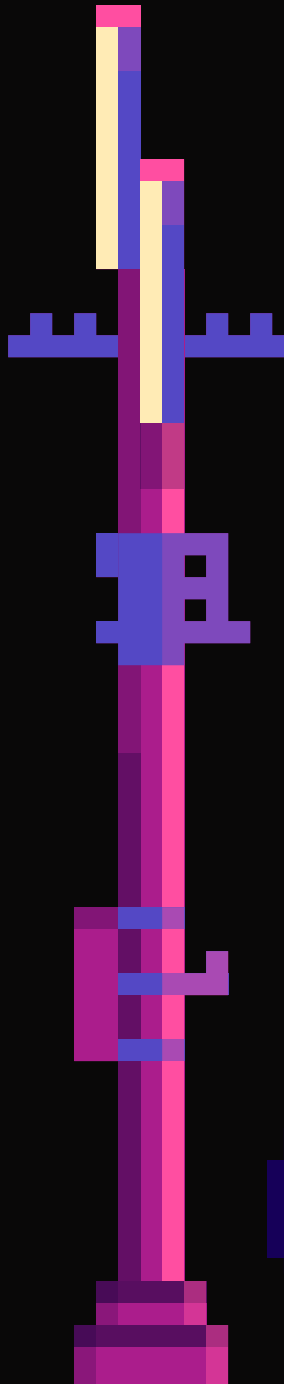




VIEW

Moves the world to simulate a camera view — like our eyes looking at the scene.

```
vec4 viewPos = viewMatrix * worldPos;
```

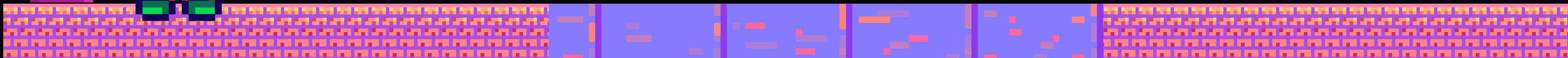
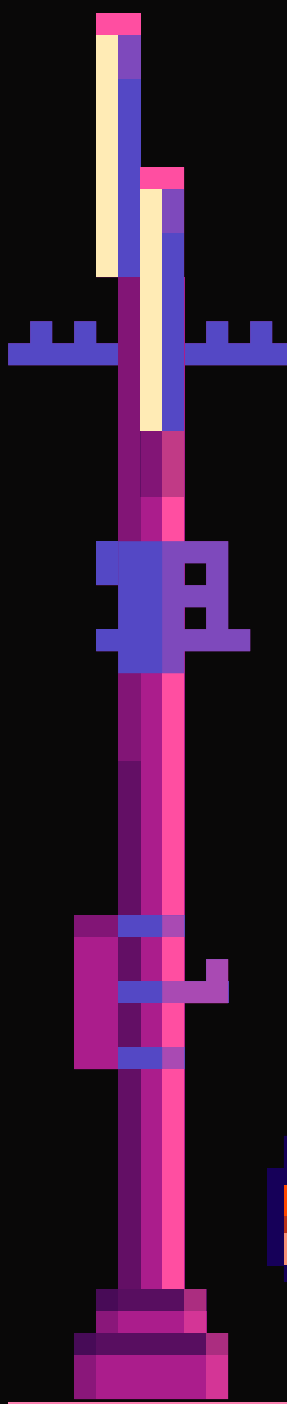




PROJECTION

Applies perspective — makes farther objects look smaller and nearby objects like bigger in size just like in real life.

```
gl_Position = projectionMatrix * viewPos;
```

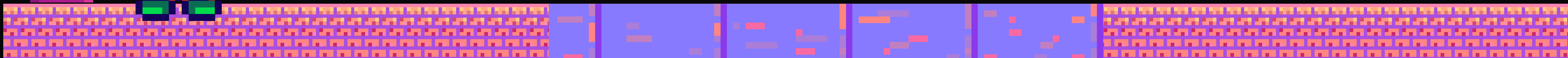
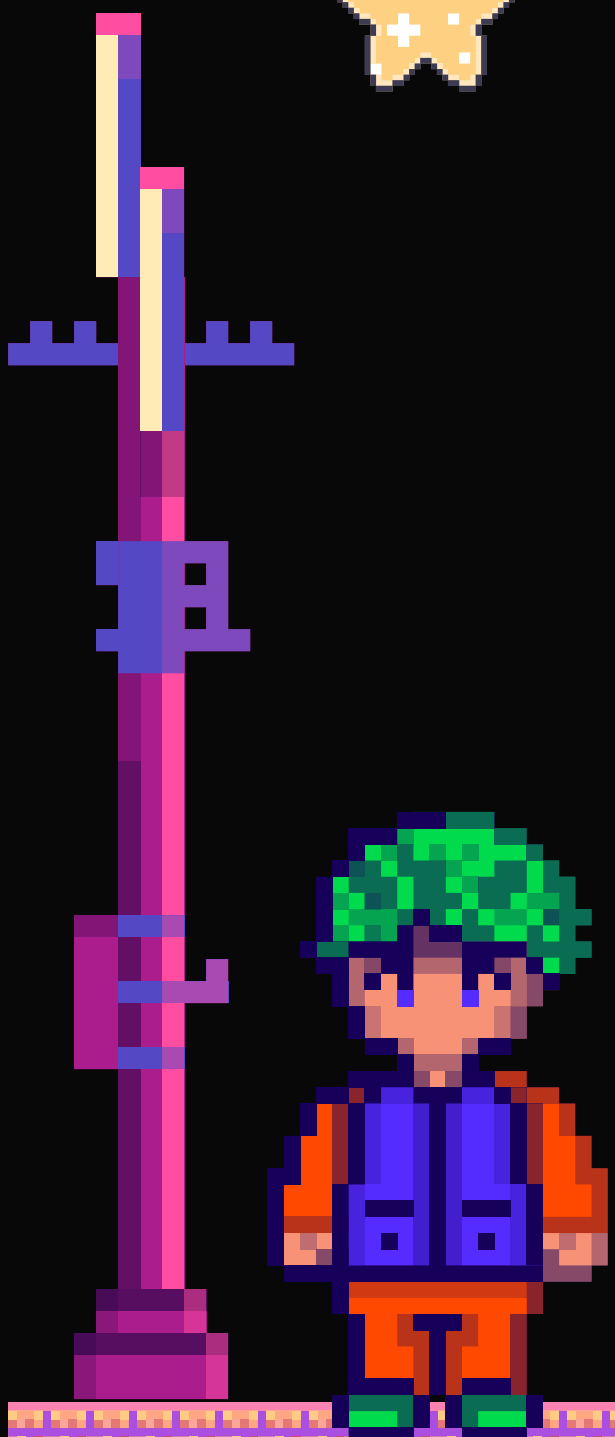




FINAL TRANSFORMATION:

Usually, this 3 are taken together for a realistic view.

```
gl_Position = projection * view * model * vec4(aPos, 1.0);
```



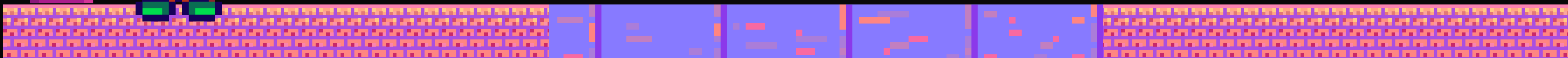


OUTPUT



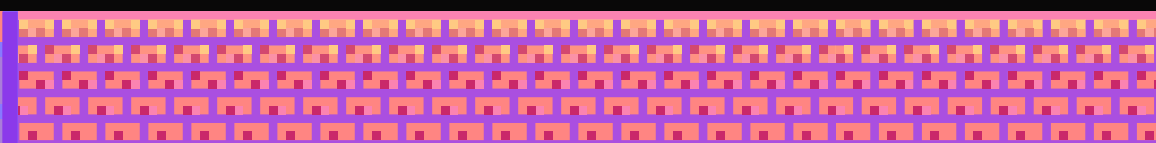
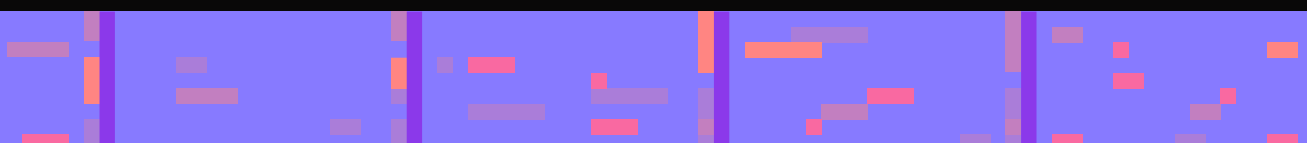
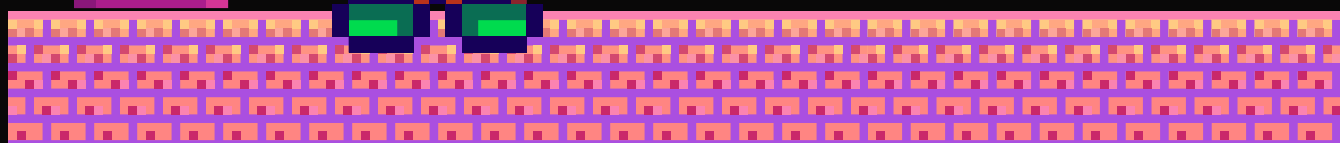
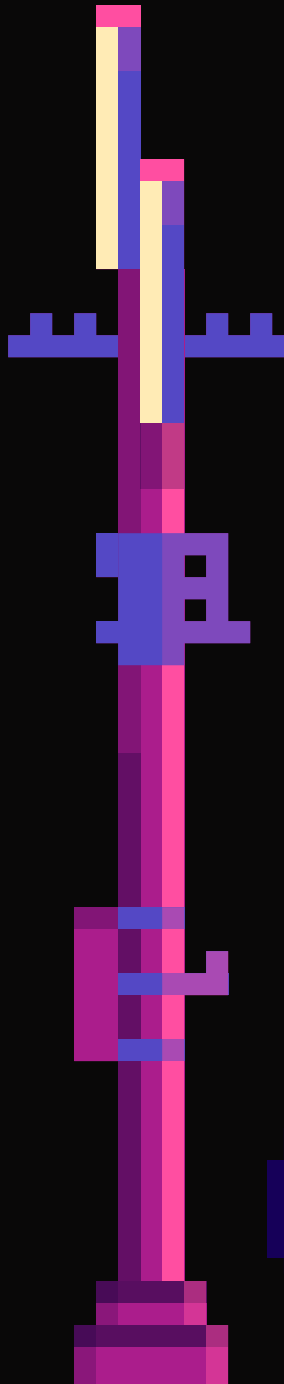
after the vertex shader does its job the output is send to the next stage of the pipeline >>>

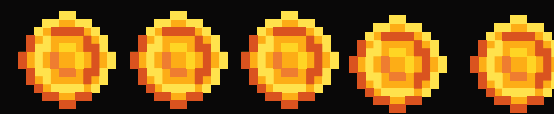
(HINT : Fragment Shader)





BUT HOW TO GIVE INPUT
TO THE VERTEX SHADER?

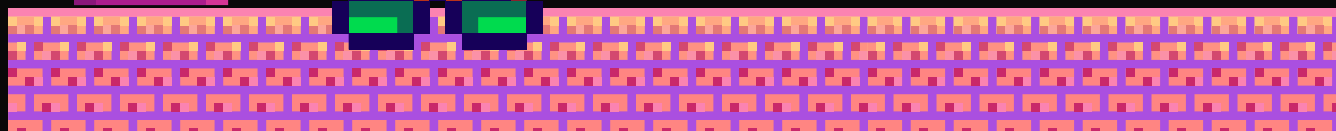




VBO



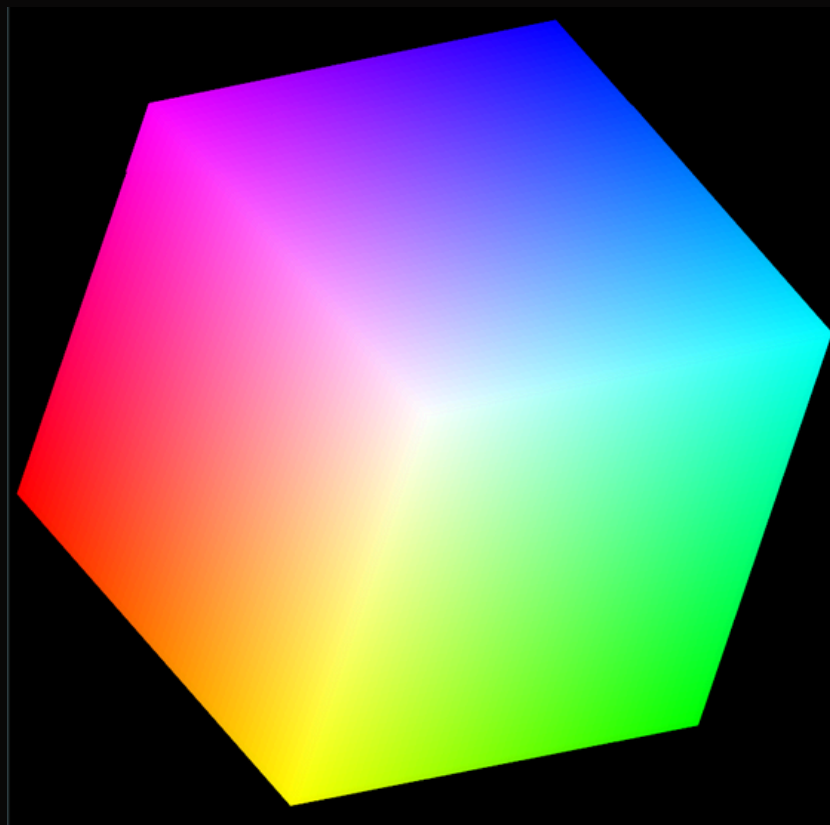
A VBO(vertex buffer object) stores the raw vertex data and then gives it to the vertex shader for processing. it is basically a chunk of memory (usually on the GPU) where vertex data like positions, colors, normals, etc., are stored. It can store the object's vertex data.





FRAGMENT SHADERS

Its job is to decide the final color of each pixel.



Takes in: output of vertex shader, textures, lighting, etc.



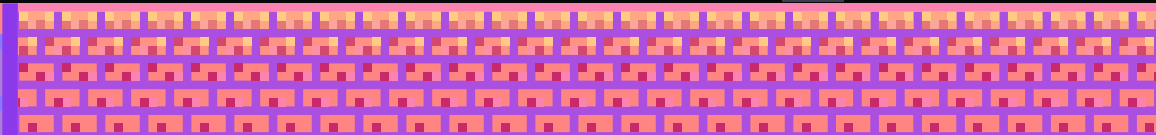
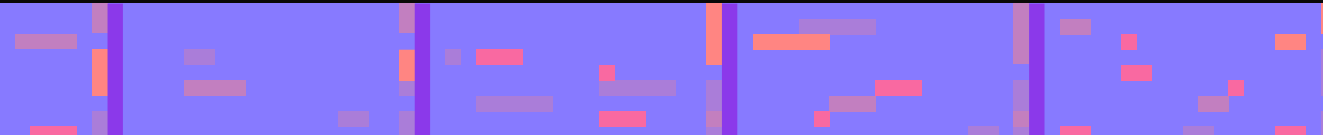
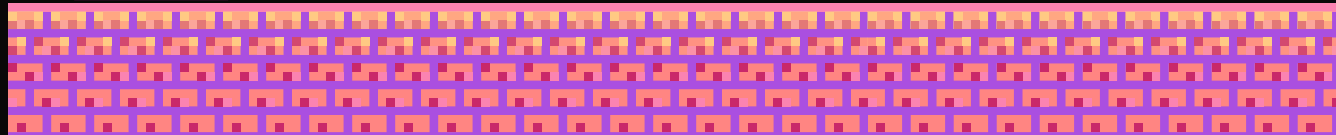
does some calculation for the final pixel colour



outputs the final pixel colour to the GPU



The GPU draws the pixel with that final colour.





INS AND OUTS

In GLSL, in and out keywords are used for passing data between shader stages



1. Vertex Shader Input (in) – Receives vertex attributes from the CPU, like position and color.



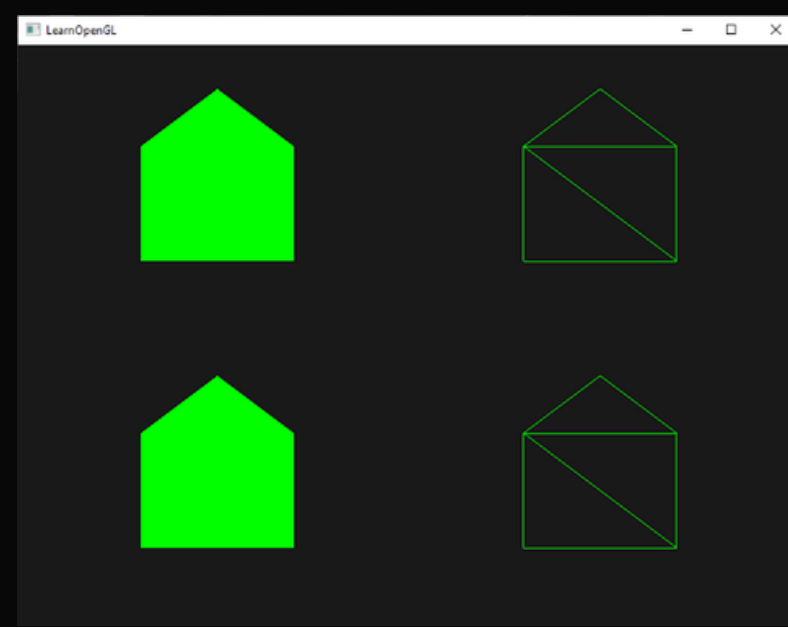
2. Vertex Shader Output (out) – Passes data to the fragment shader.
Example:



3. Fragment Shader Input (in) – Receives interpolated data from the vertex shader.



4. Fragment Shader Output (out) – Specifies the final pixel color.





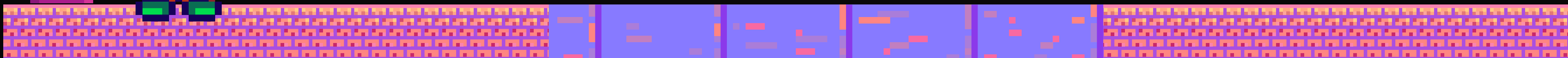
SHADER PROGRAM



the vertex shader and fragment shader together comes under
the shader program.

YOU THEN TELL OPENGL:

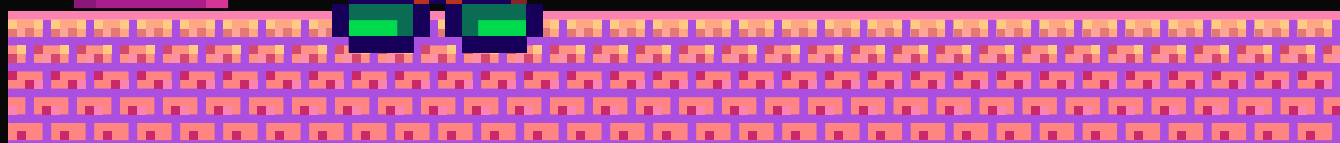
☞ "HEY, USE THIS SHADER PROGRAM TO DRAW MY
OBJECT!"





A Shader Program is like a set of instructions given to the GPU to decide:

- Where to place the shapes (Vertex Shader)
- What color each pixel should be (Fragment Shader)



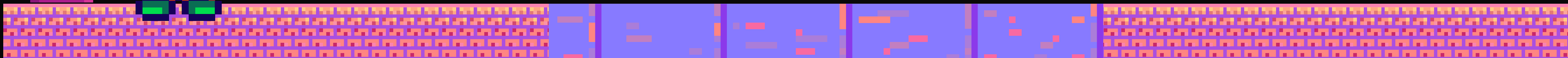
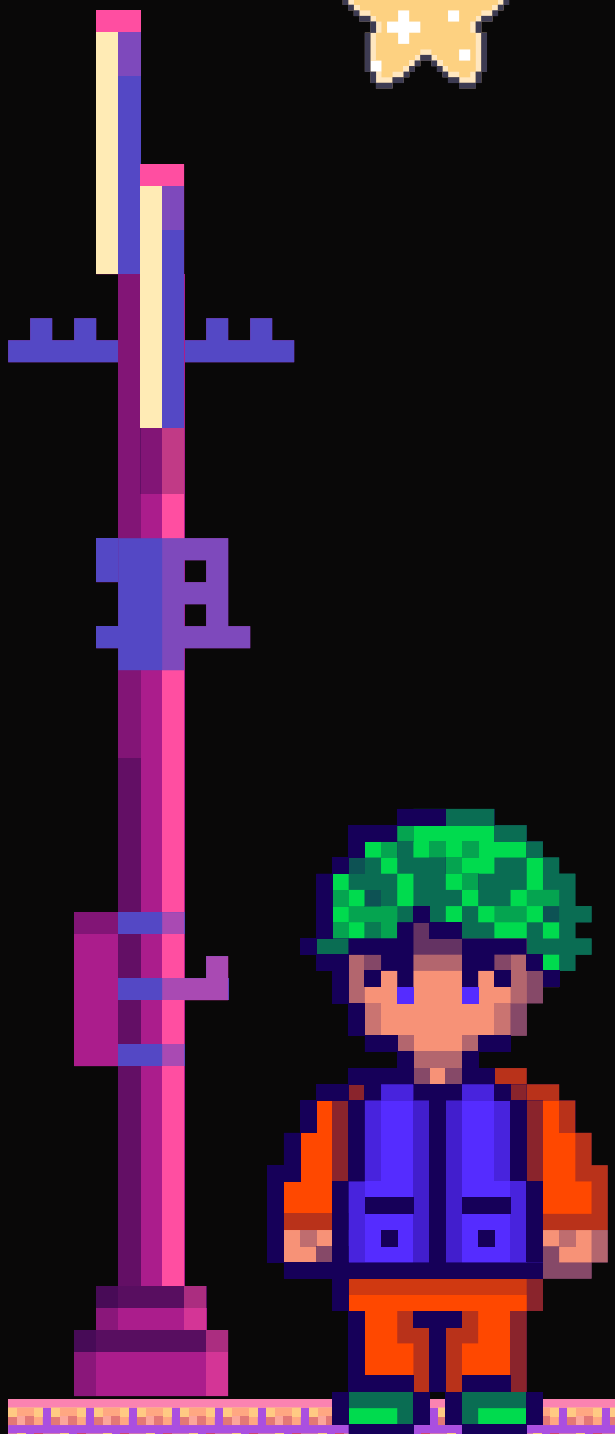


firstly, compile the shaders ...



```
// Vertex shader
unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);

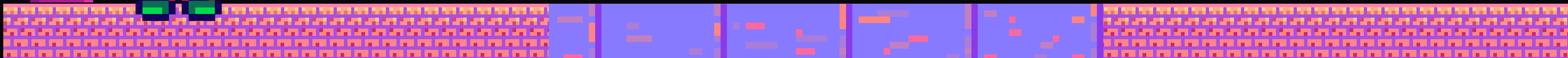
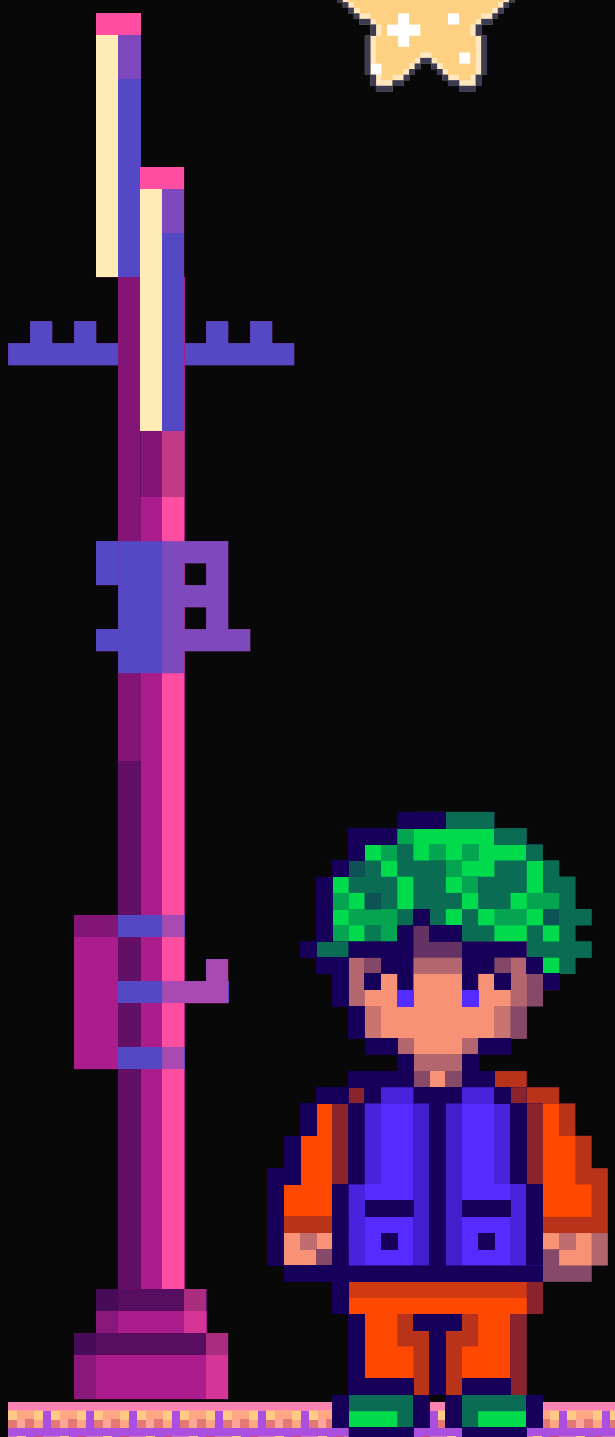
// Fragment shader
unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
```





then, link them to the shader
program

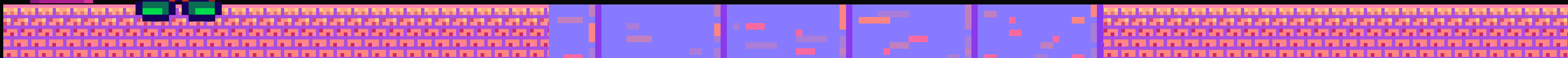
```
unsigned int shaderProgram = glCreateProgram();  
glAttachShader(shaderProgram, vertexShader);  
glAttachShader(shaderProgram, fragmentShader);  
glLinkProgram(shaderProgram);
```





after compilation and linking we can finally use the shader program by just writing:

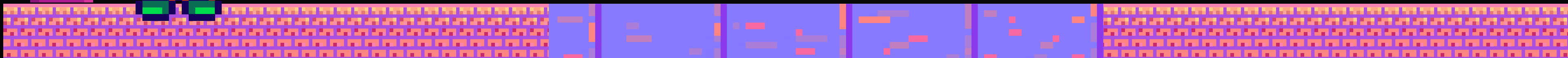
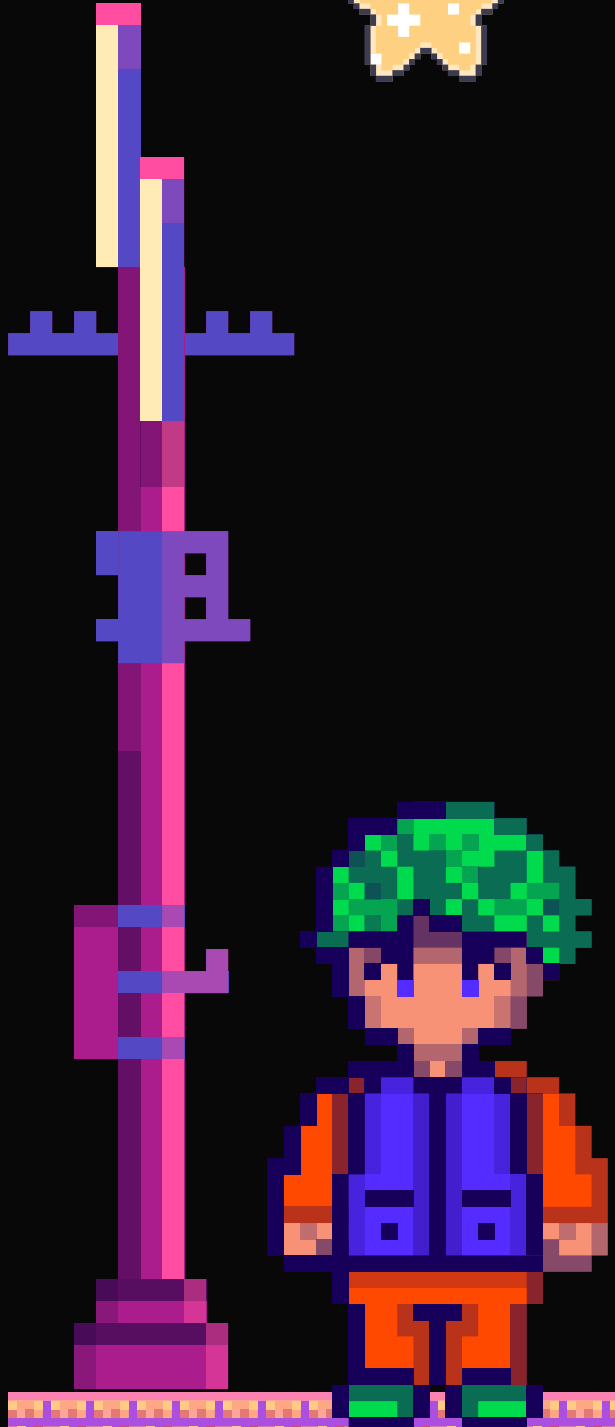
```
glUseProgram(shaderProgram);
```





And...
Don't forget to clean the environment.

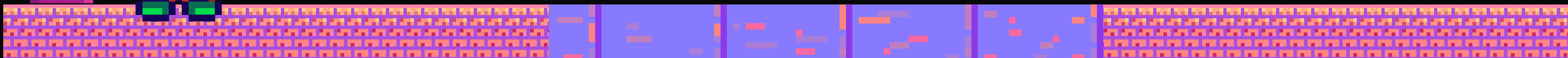
```
glDeleteShader(vertexShader);  
glDeleteShader(fragmentShader);
```





WAIT !

Between the vertex shader and the fragment shader comes the geometry pipeline there can be another optional shader which is the geometry shader that takes the vertex shader output, does something fishy and then returns it to the fragment shader.





GEOMETRY SHADER



takes the shapes made of points, lines, or triangles from the Vertex Shader and can create new shapes or modify existing ones before they go further



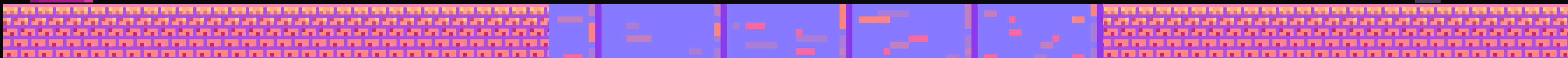
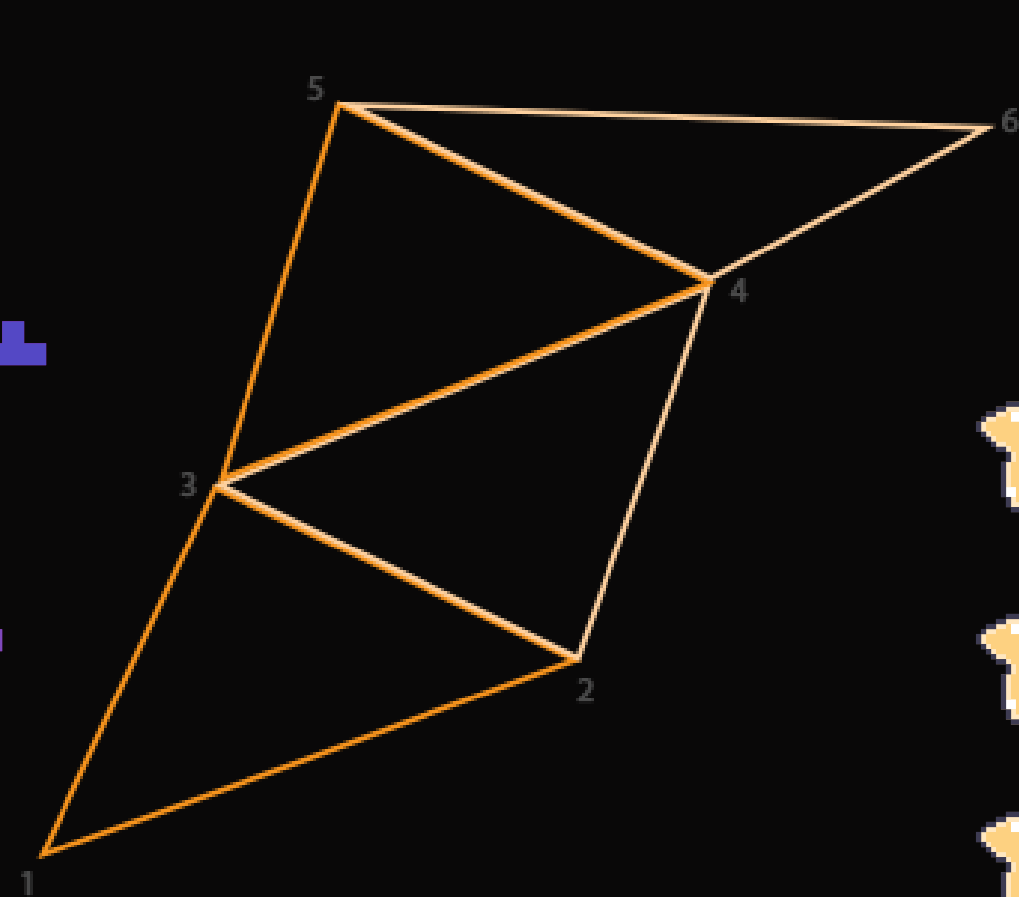
adds extra details to models (by creating more triangles (more geometry))



can calculate new data like normals, directions or additional coordinates



can create outpines, wireframes and other cool effects

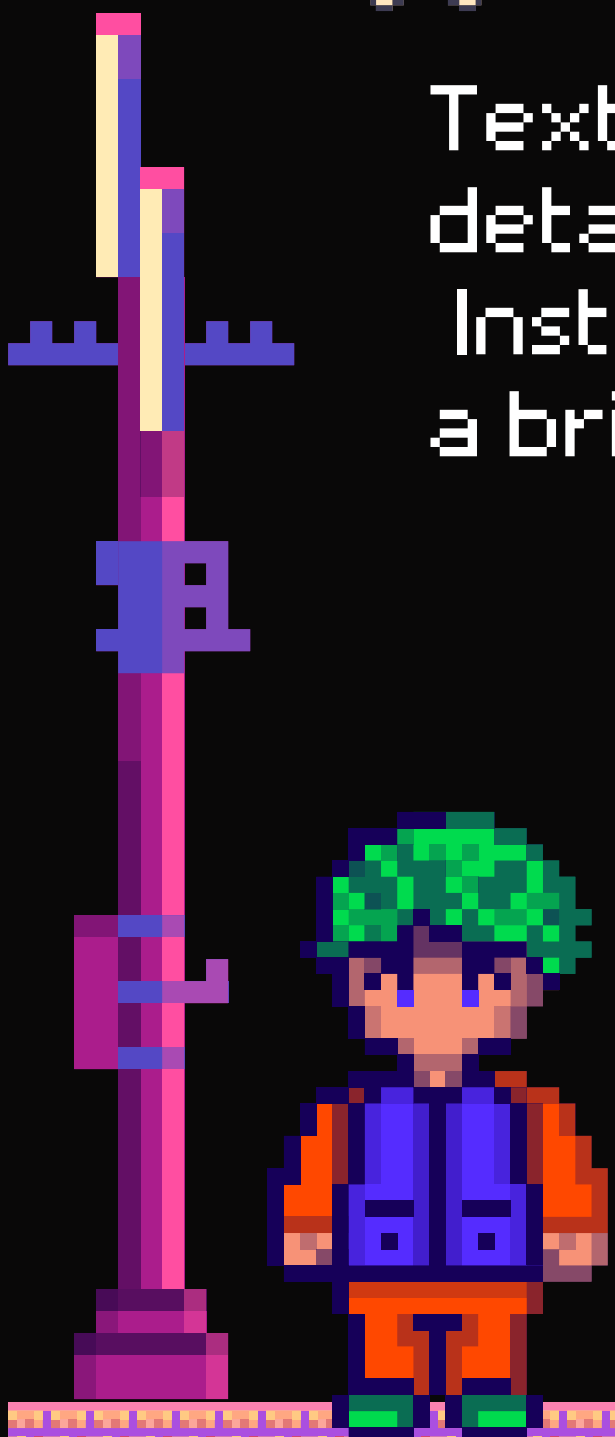
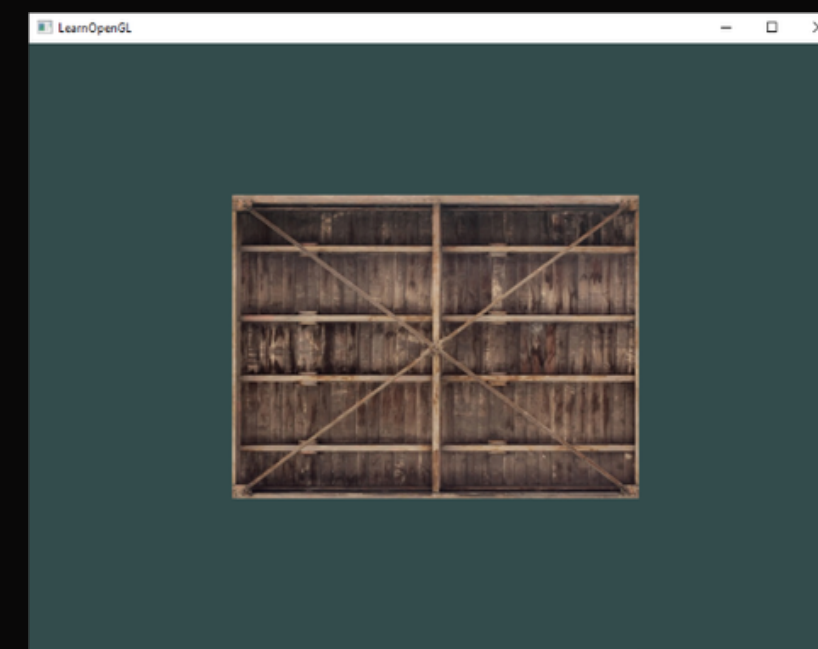
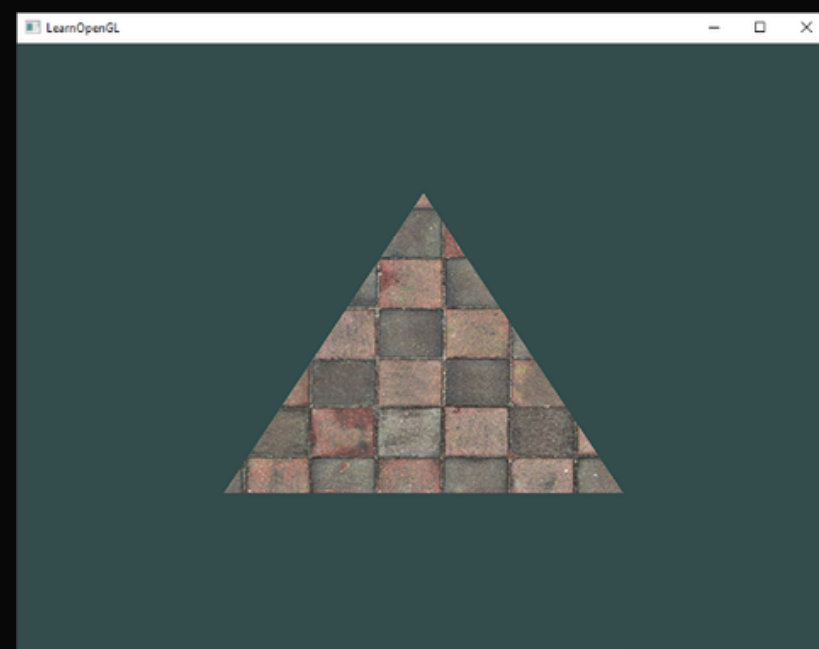




TEXTURES

Textures are images that are mapped onto 3D objects to give them detail and realism.

Instead of coloring a shape with a flat color, we wrap an image (like a brick wall or wood pattern) over it.

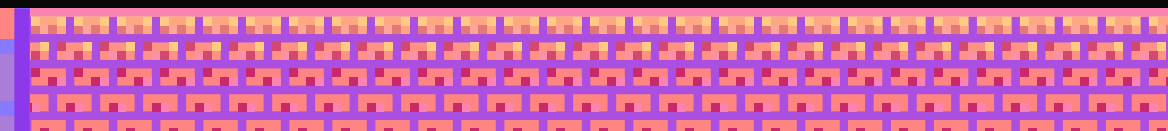
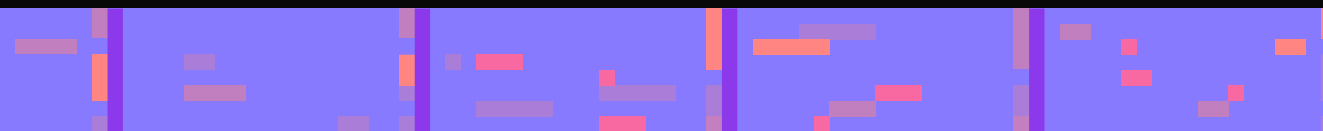
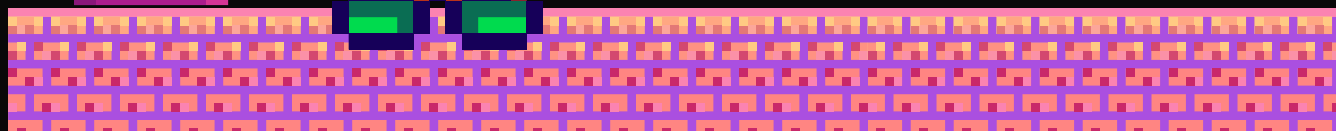
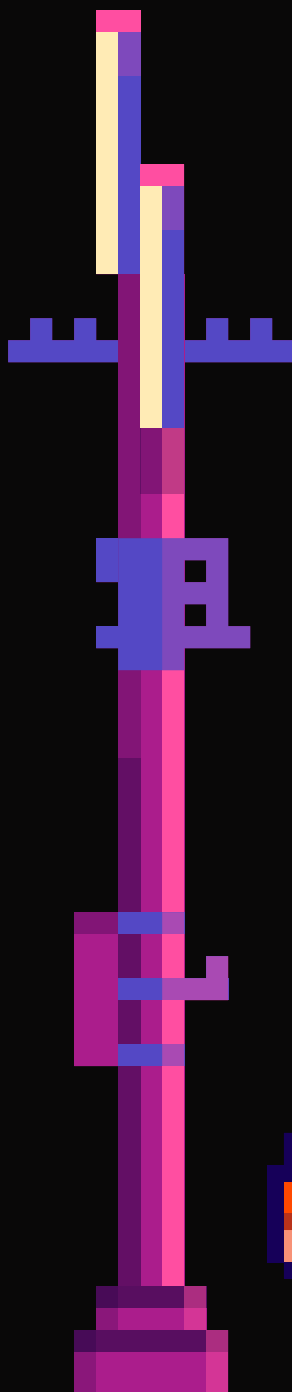




CAMERA

In OpenGL, a camera doesn't exist as a real object — instead, we move the world around the camera to simulate camera movement.

It defines what part of the 3D world we see and from what angle.

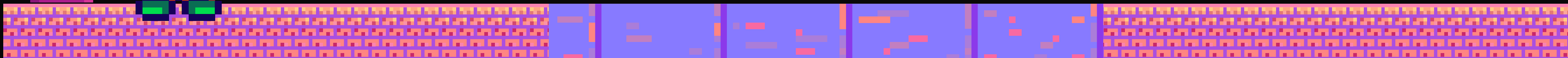
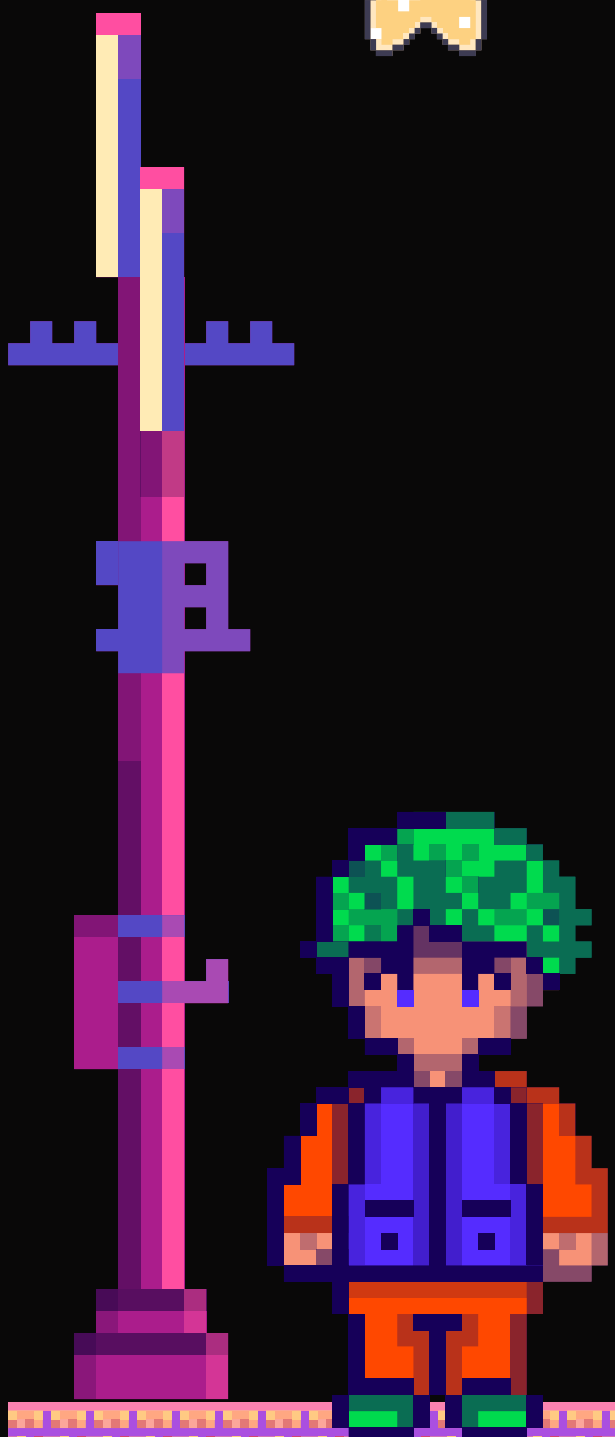
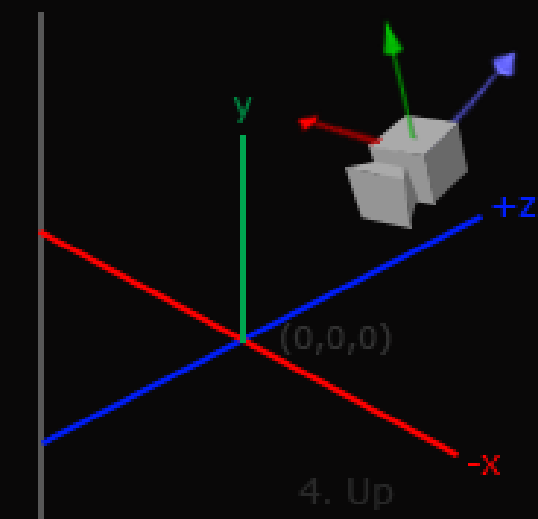
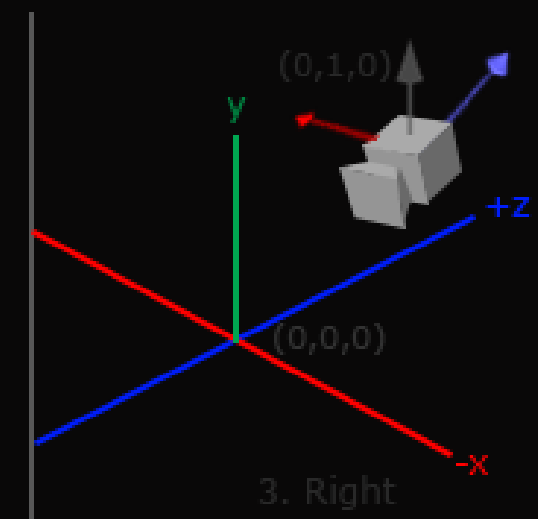
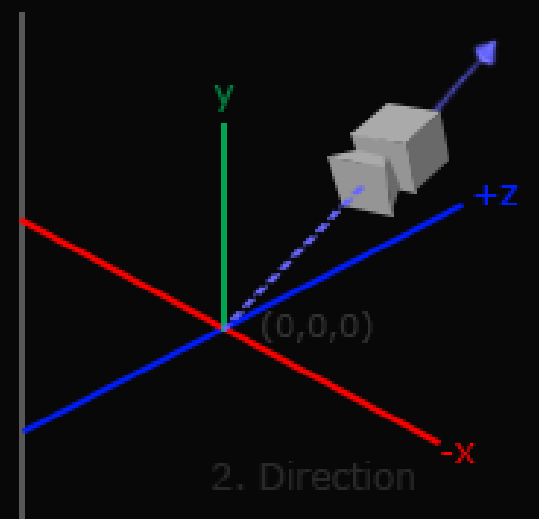
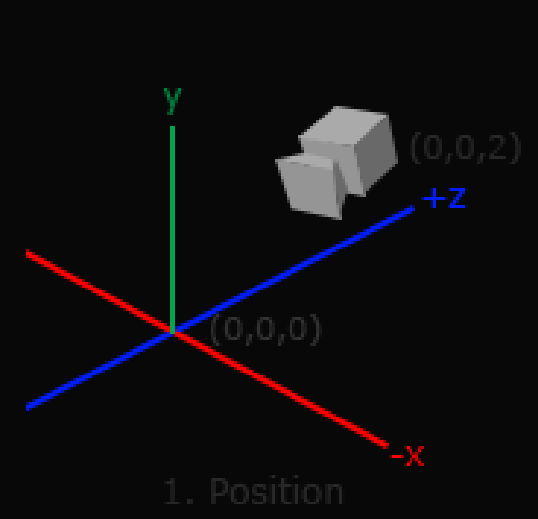




INPUTS REQUIRED !



- Position: Where the camera is.
- Target: To which direction the camera looks at.
- Right axis: positive x-axis of the camera space
- Up axis: Defines the "up" direction.





I. CAMERA POSITION

The camera position is simply a vector in world space that points to the camera's position. Given by:

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
```

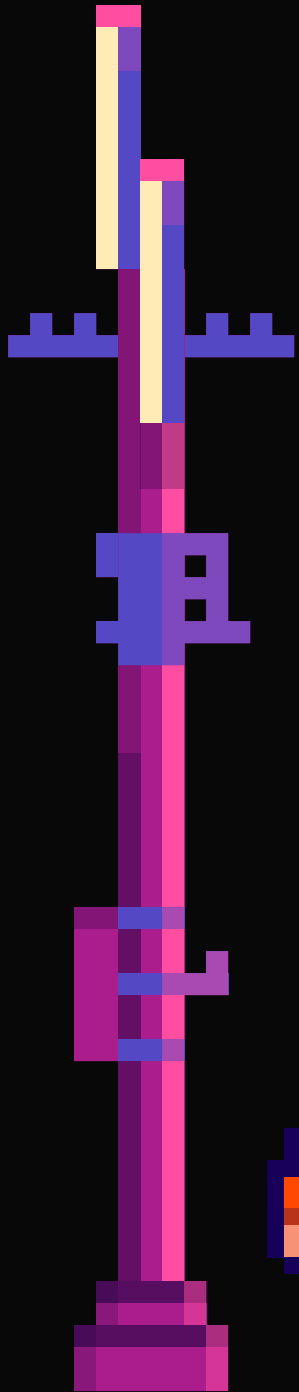




II. CAMERA DIRECTION

Here, we have two things. One, the camera, other, the target at which the camera is looking. Let's take target at origin (0,0,0). Camera direction is given by:

```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);  
glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);
```





III. RIGHT VECTOR

This is the positive x-axis of the camera space. We simply take any up vector that points upwards in space and take its cross product with the camera's z-vector we got in step II.

```
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);  
glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
```





IV. UP VECTOR

As we already have the camera's x-axis and z-axis vector, The y-axis vector is simply cross product of both axes.

```
glm::vec3 cameraUp = glm::cross(cameraDirection, cameraRight);
```



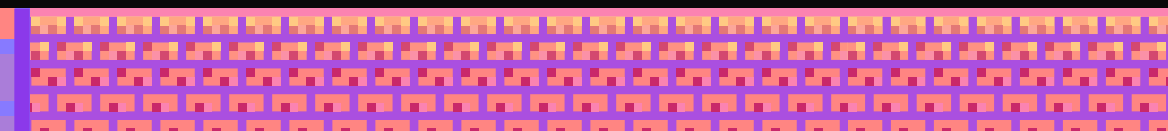
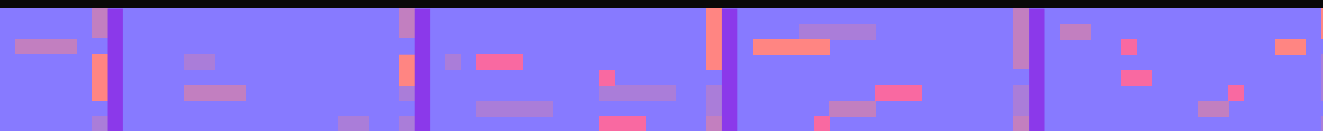
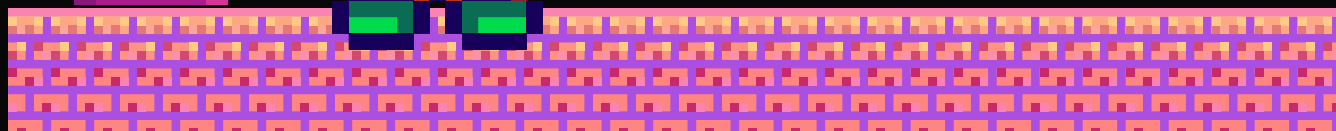
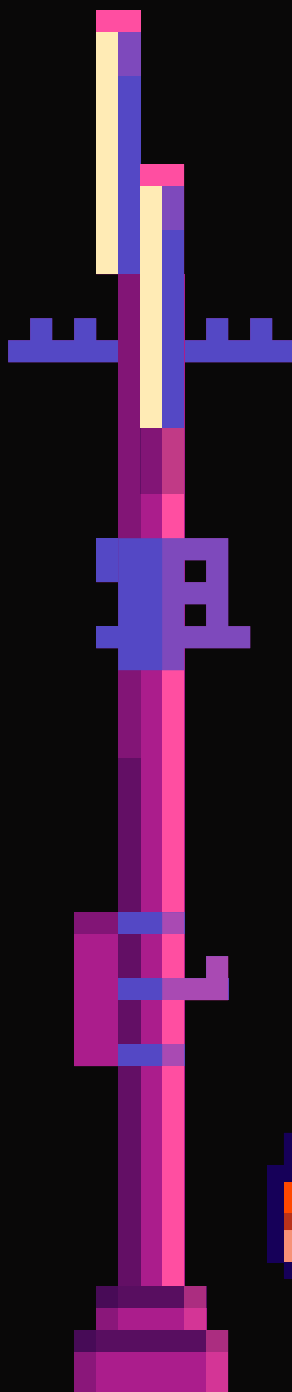


SIMPLE LIGHTING




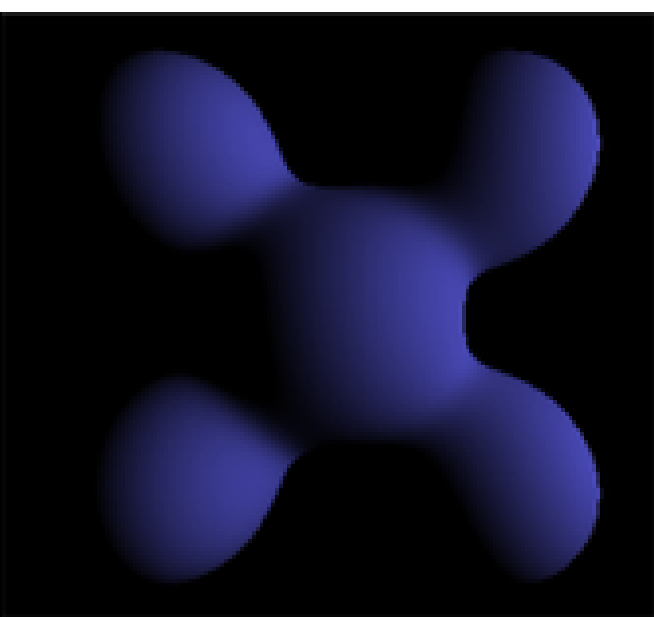
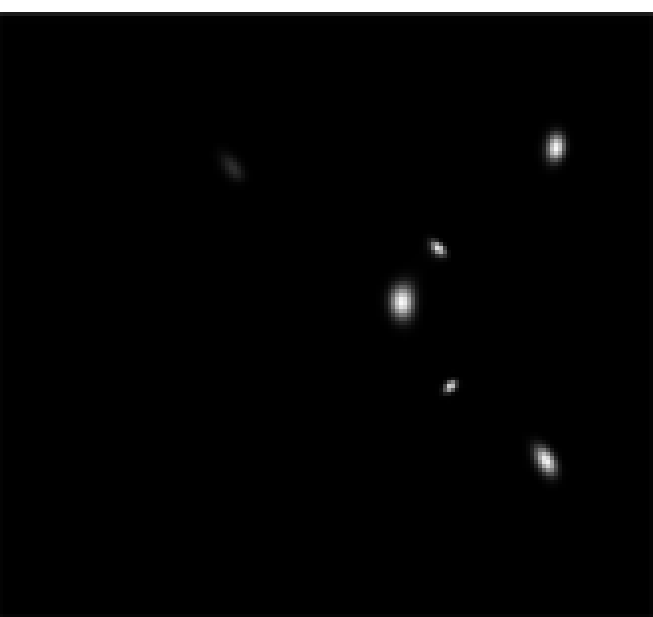
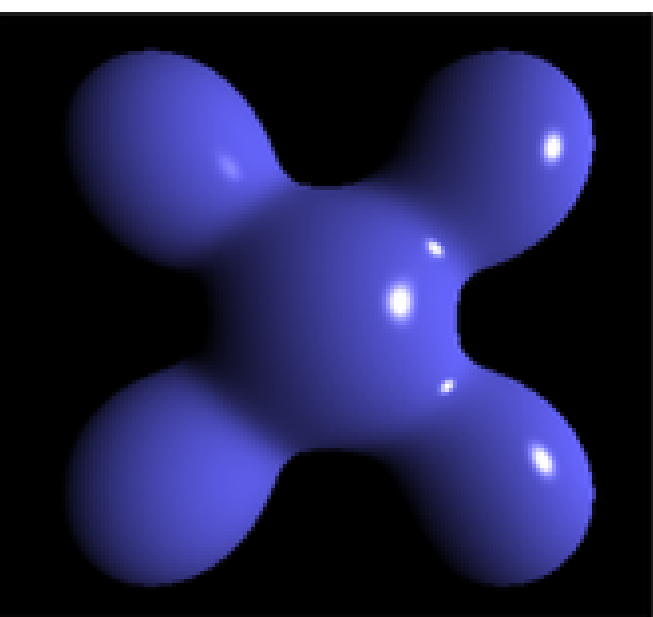
Lighting helps us simulate real-world light effects like brightness, shadows, and highlights. It is a very important part while creating a 3D scene. It has the power to change the render results.

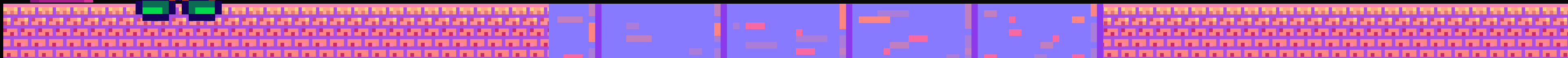
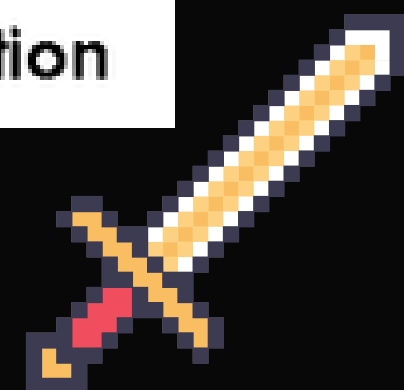
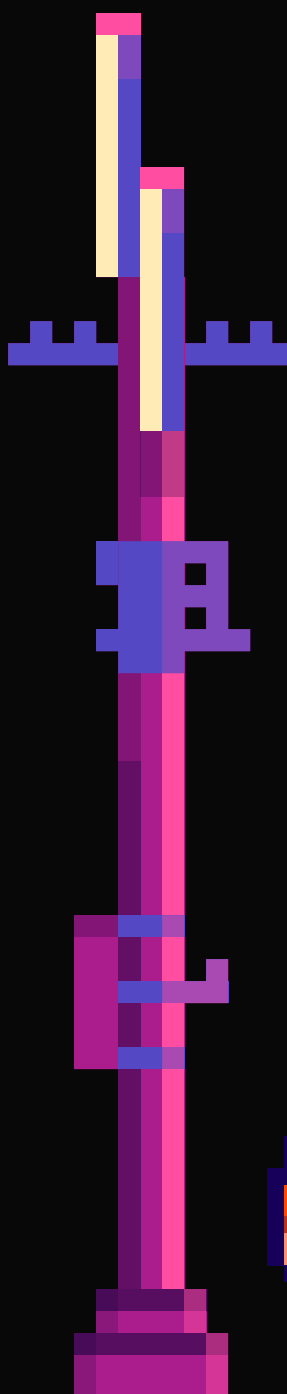
A famous lighting model is the Blinn-Phong model. It is a popular lighting technique used in computer graphics to simulate realistic lighting on 3D surfaces.





PHONG LIGHTING MODEL

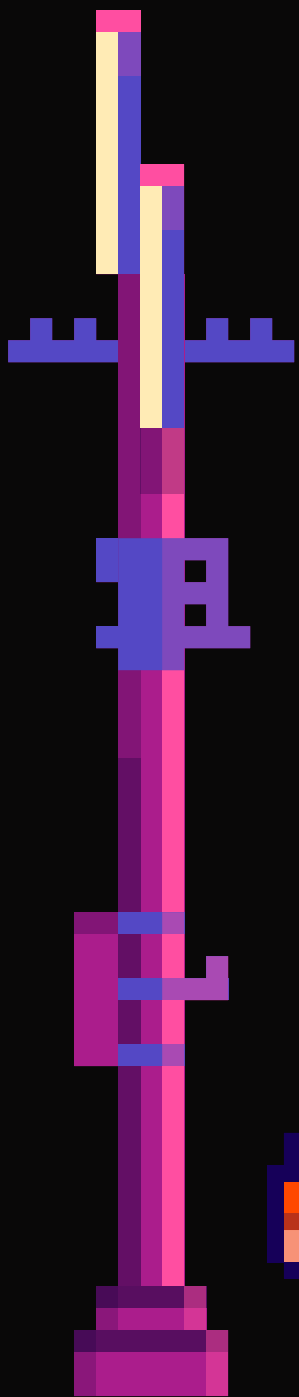
	+		+		=	
Ambient		Diffuse		Specular		Phong Reflection





INPUTS REQUIRED !

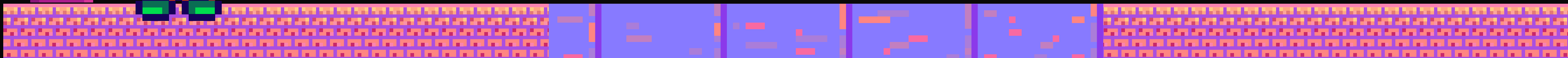
- Normal vector: tells which direction the surface is facing.
- Light position: where the light source is in the scene.
- View direction: from the camera to the object.
- Material properties: like shininess or color, textures.
- Light color and intensity.





FURTHER ...

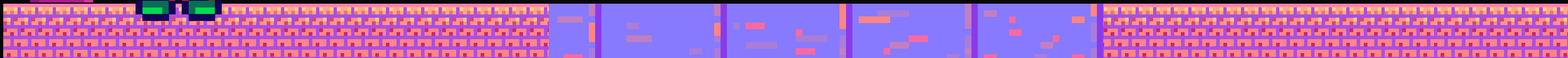
(TYPES)





I. AMBIENT LIGHTING

General light that's everywhere in the scene even when there is no specific light source of direction of light. So, objects are not completely dark. Like the moon light.

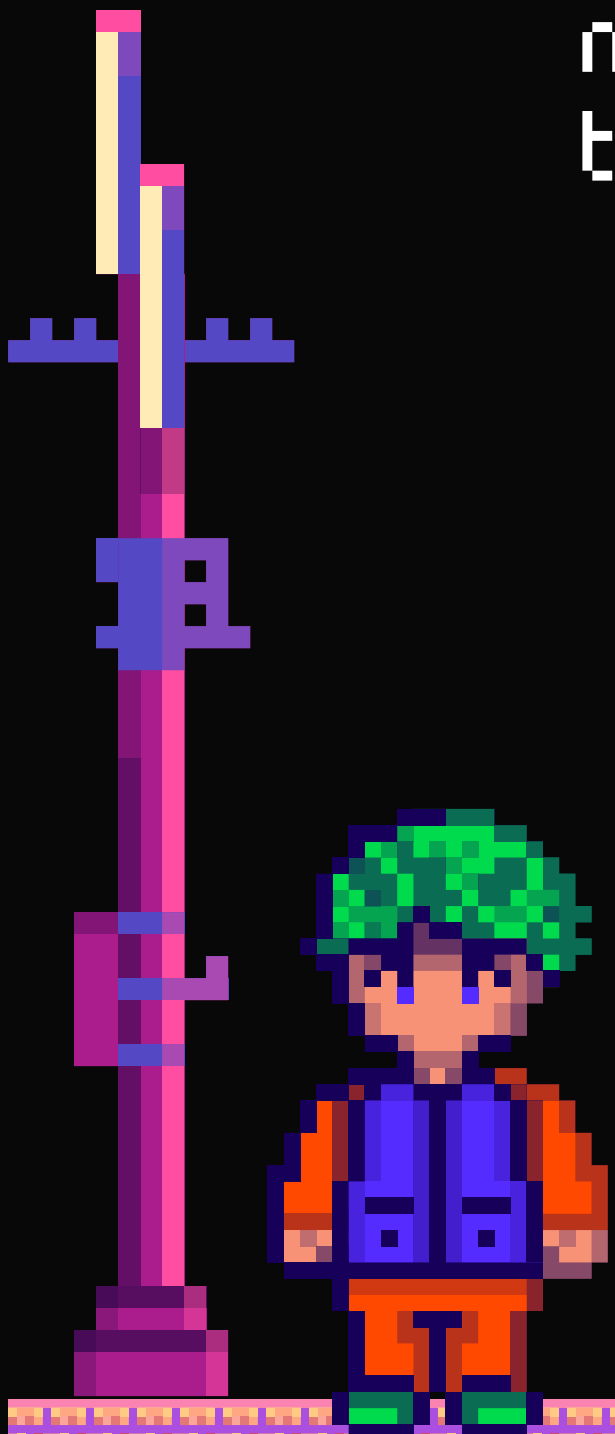




Adding ambient lighting to the scene is really easy. We just multiply the light colour and an ambient strength constant to the object's colour.

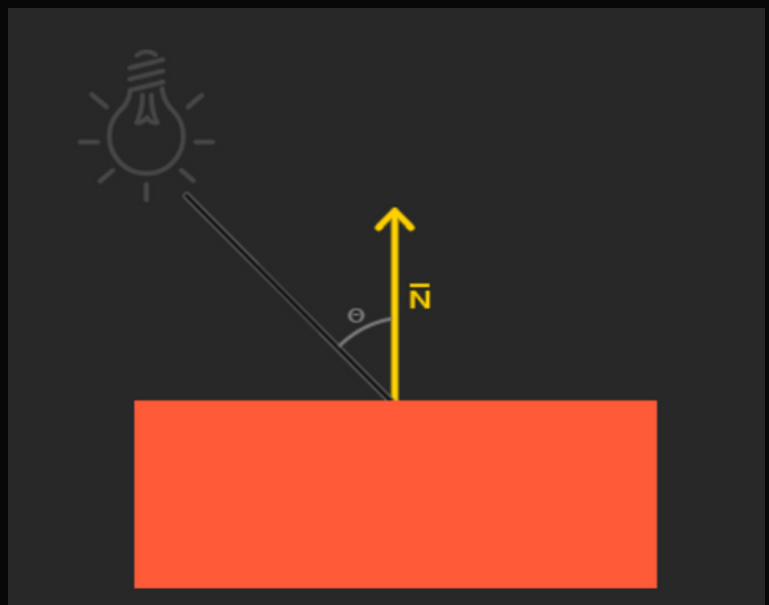
```
void main()
{
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    vec3 result = ambient * objectColor;
    FragColor = vec4(result, 1.0);
}
```



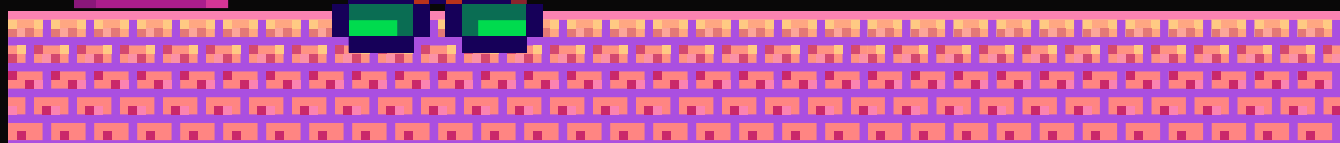
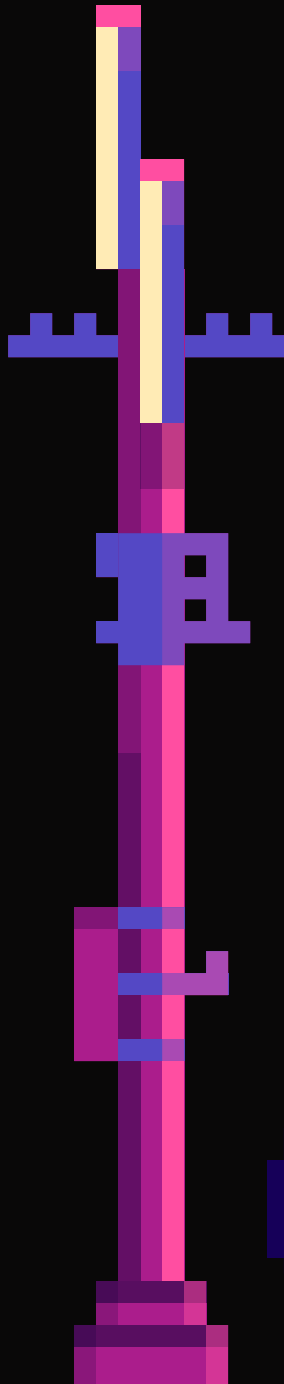


II. DIFFUSE LIGHTING



DIFFUSE LIGHTING IS THE SOFT LIGHT THAT SPREADS ACROSS A SURFACE WHEN IT FACES A LIGHT SOURCE, MAKING IT LOOK BRIGHT ON THE SIDE FACING THE LIGHT.

Diffuse lighting simulates how light spreads evenly over a surface, depending on the angle between the light direction and the surface's normal.





SOME MATHS . . .

```
float diff = max(dot(norm, lightDir), 0.0);  
vec3 diffuse = diff * lightColor;
```

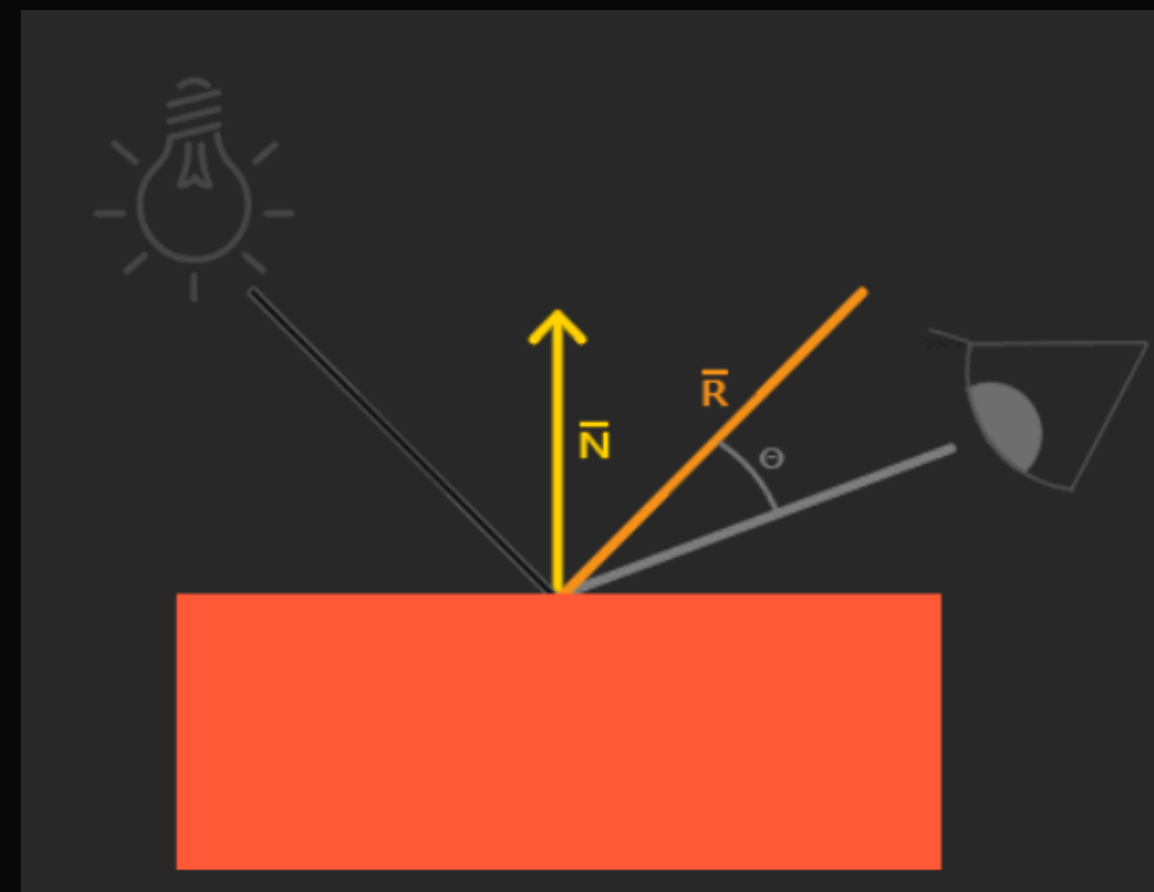
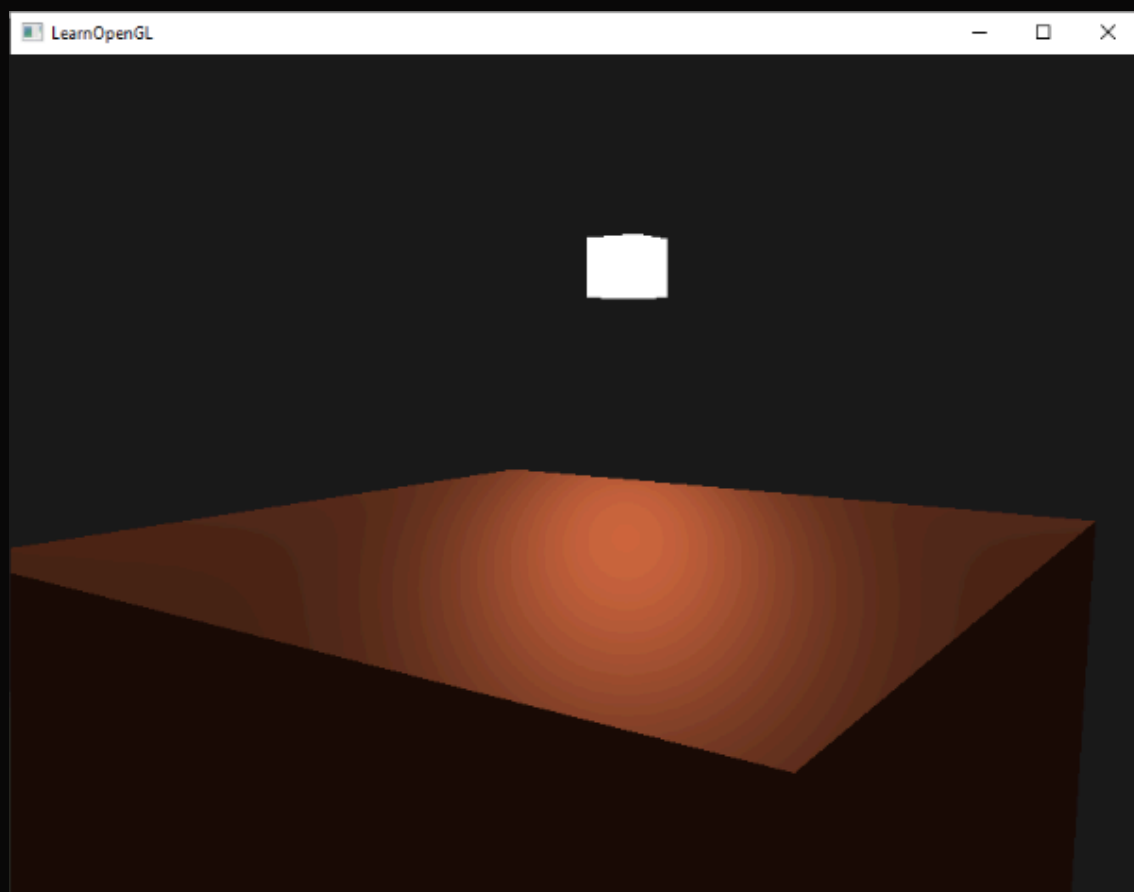
where `diff` is a factor that depends on the dot product of light and surface normal vector. the higher the value the more lit it is. Due to dot product, the more the light direct along the normal the brighter that side of the object looks (as expected). Hence, the final object color outputed is:

```
diffuse = lightColor × objectColor × max(dot(norm, lightDir), 0.0)
```





III. SPECULAR LIGHTING



Similar to diffuse lighting, but also accounts the direction from where the viewer sees the fragment.

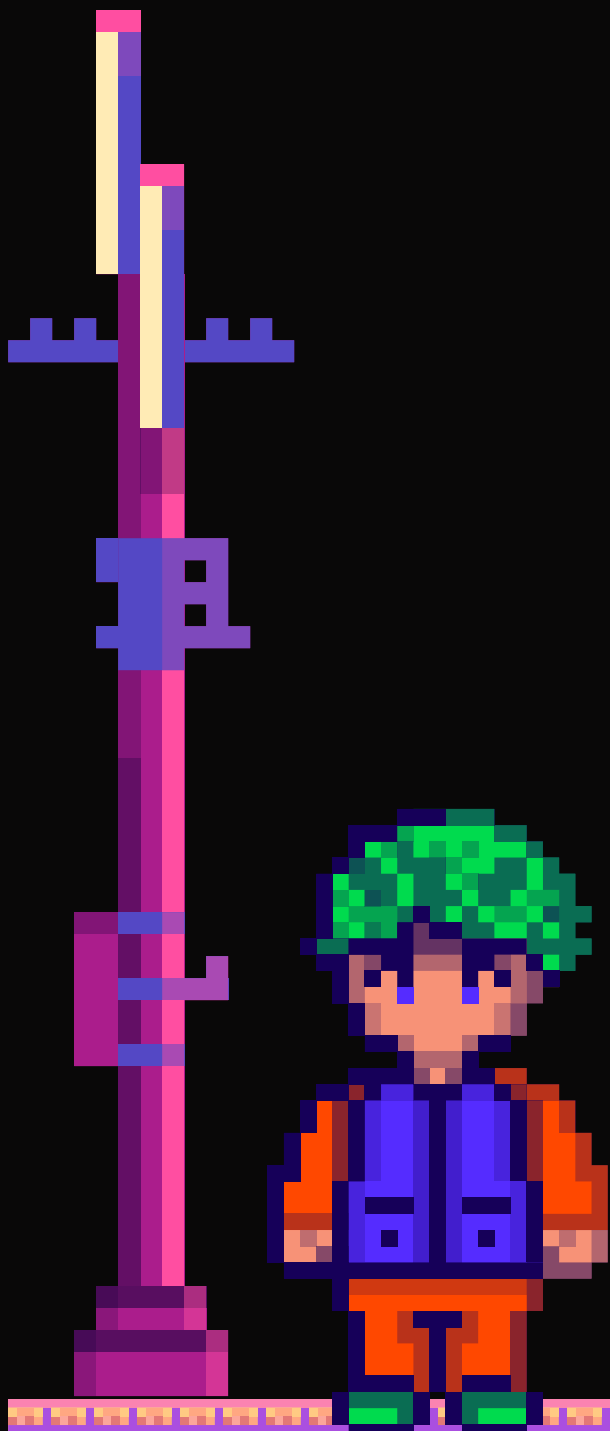




```
vec3 lightDir = normalize(lightPos - fragPos);    // Direction to the light
vec3 viewDir  = normalize(viewPos - fragPos);    // Direction to the camera
vec3 halfwayDir = normalize(lightDir + viewDir);  // Midpoint direction
float spec = pow(max(dot(norm, halfwayDir), 0.0), shininess);
vec3 specular = lightColor * spec;
```

Here, fragpos is the position coordinate of the fragment.
halfwayDir is the direction between light and our view
direction.

Power of light is exponent of this direction to the power
shininess. The more the shininess of the object, sharper
the bright spot.



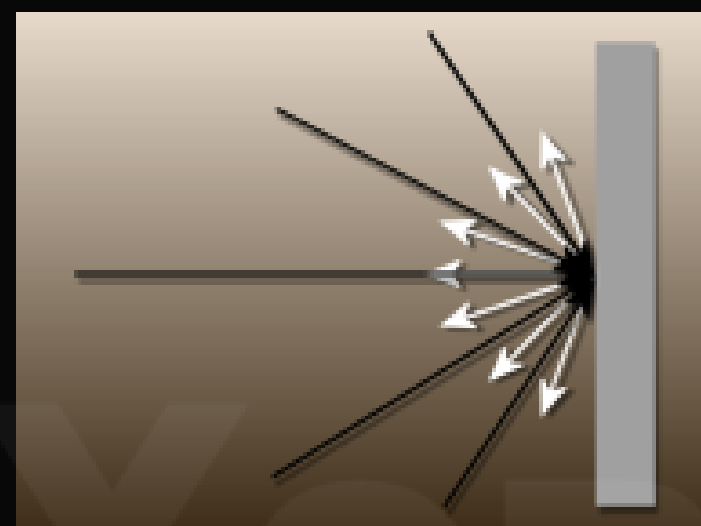


Basically,

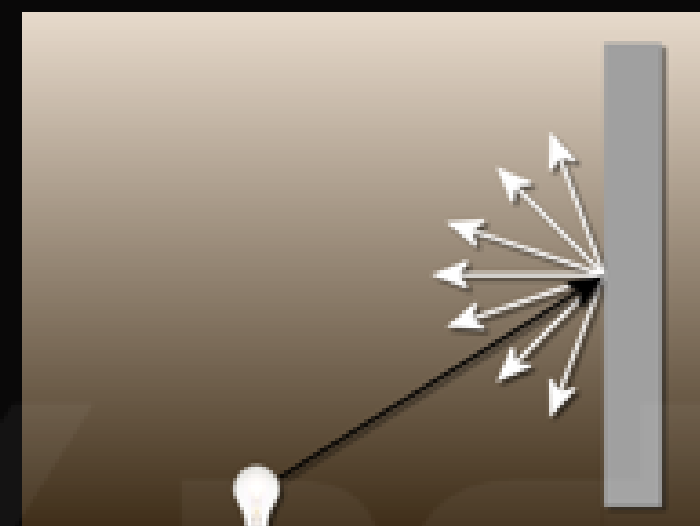


ambient

Ambient

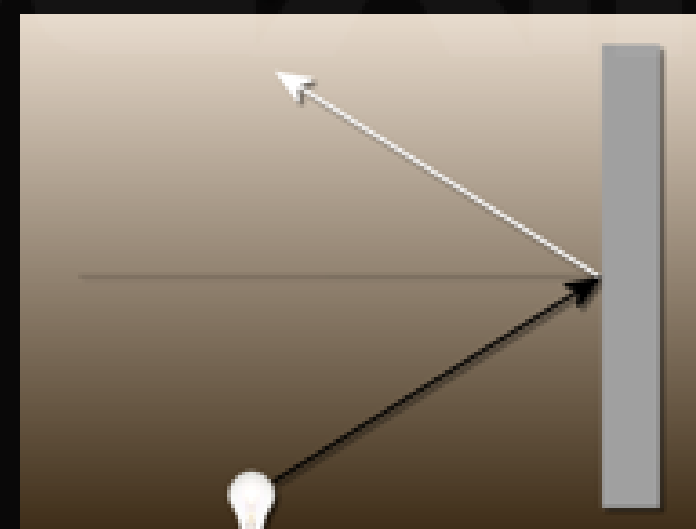


Diffuse

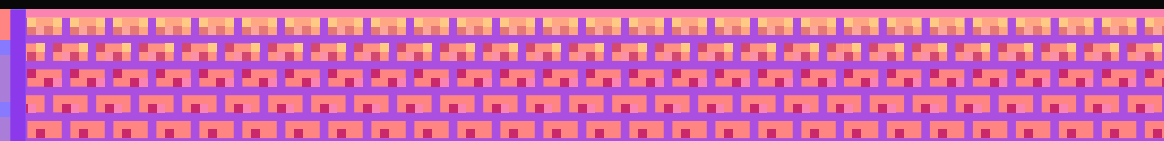
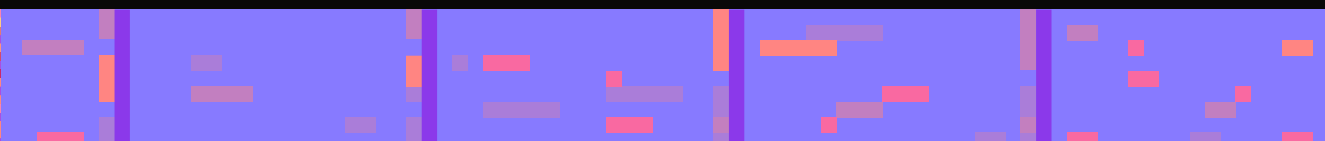
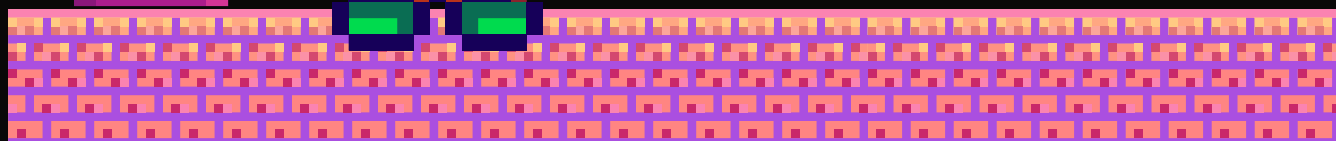
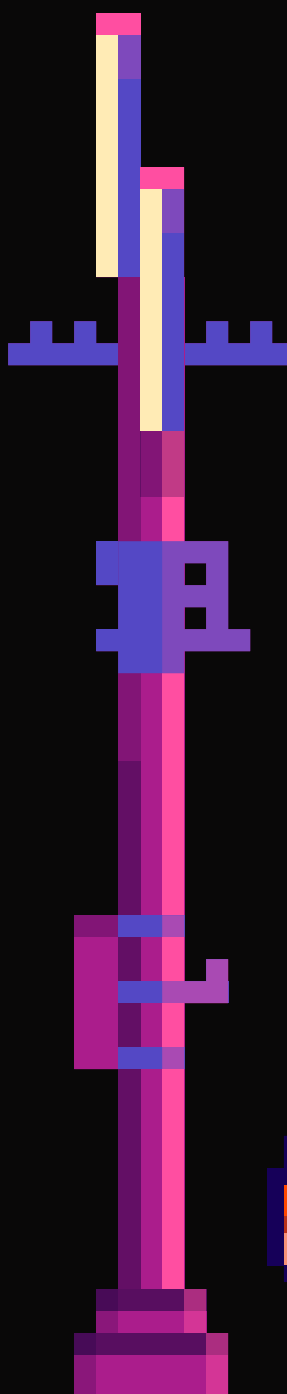


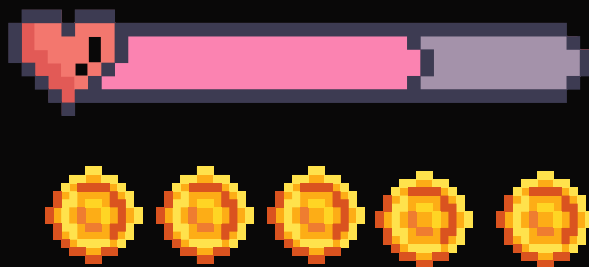
diffuse

Specular



specular



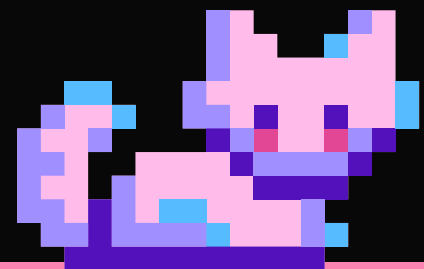
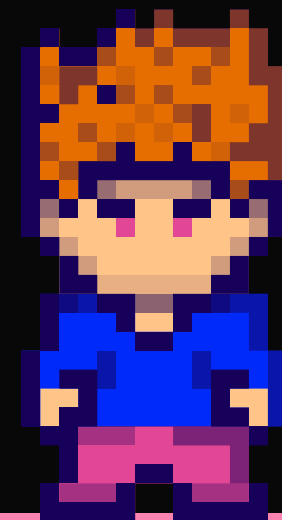
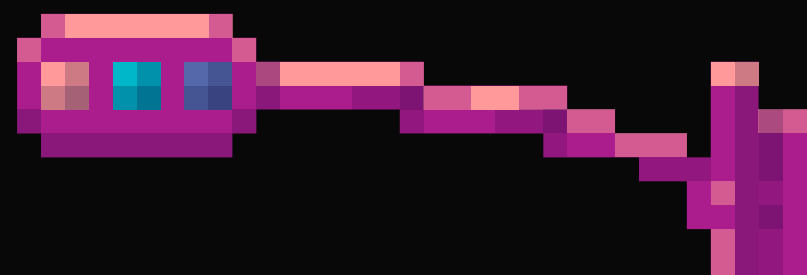


MENU

START



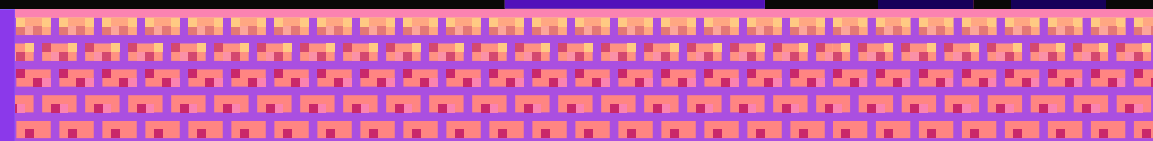
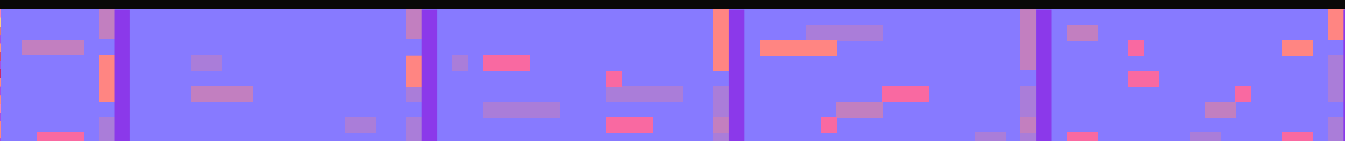
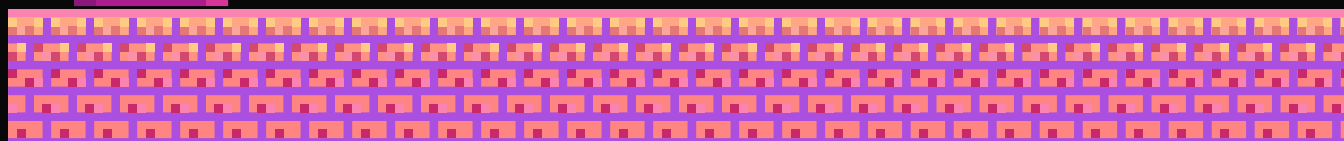
HQA





WHAT DOES A HDR DO ?

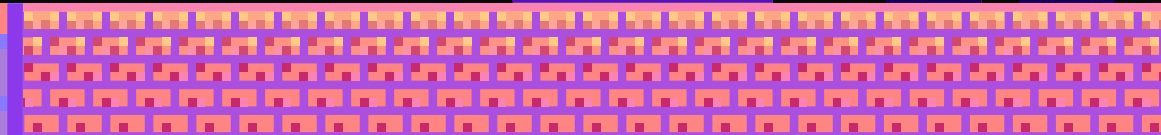
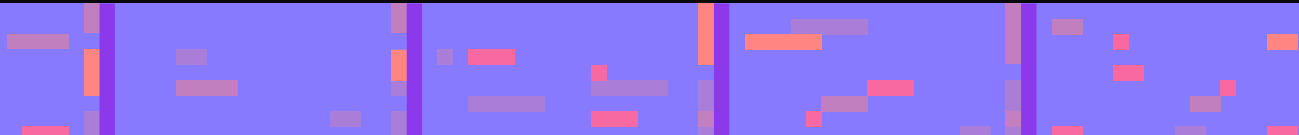
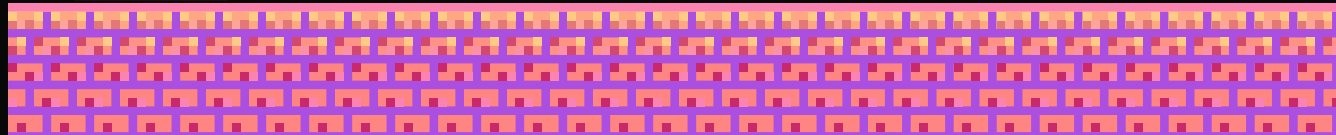
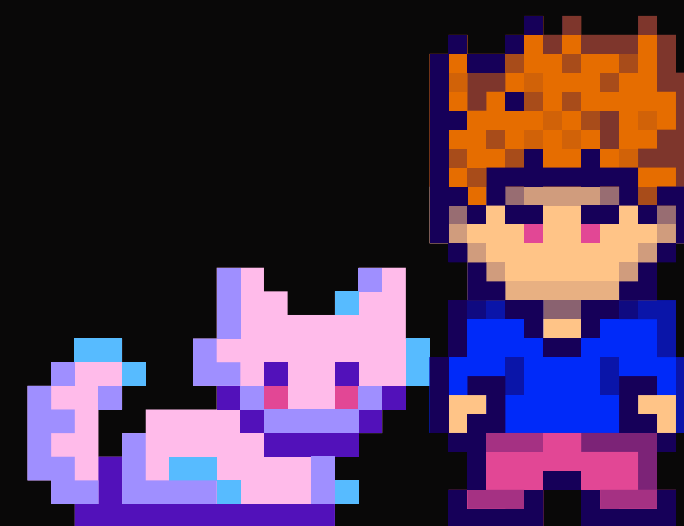
HDR lets us use brightness levels higher than normal, so bright things
can really shine, dark areas stay dark, and we don't lose small
details—even in very bright or dark scenes.





GOOD TO KNOW !

Brightness and color values, by default, are clamped between 0.0 and 1.0 when stored into a framebuffer (LDR). LDR stands for low dynamic range it clamps the brightness value of colors between 0 and 1.

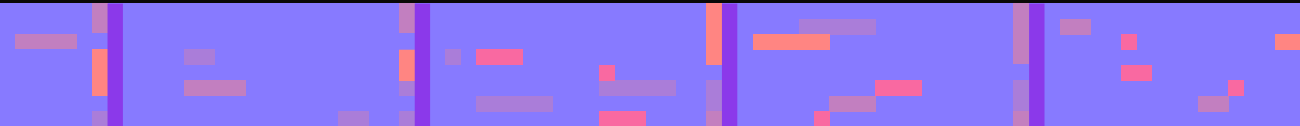
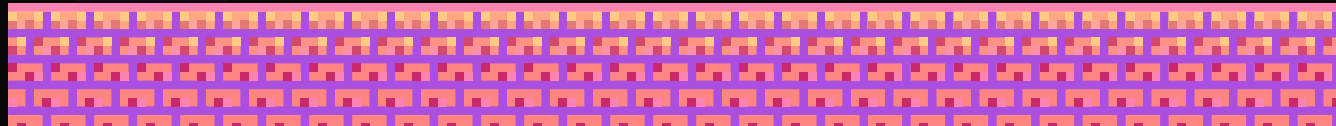
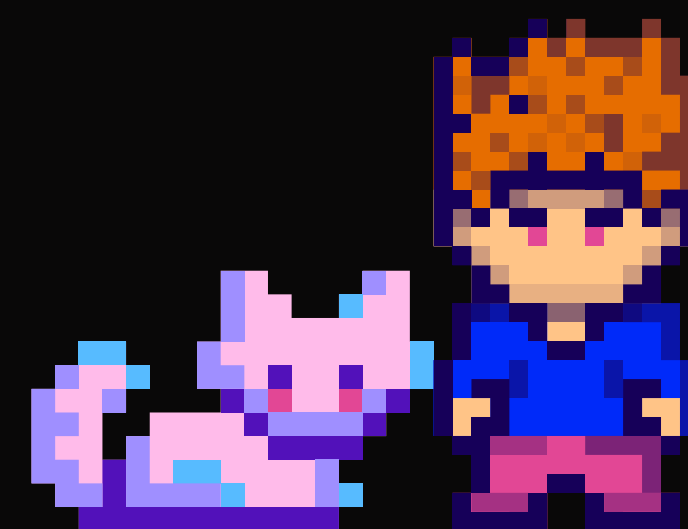
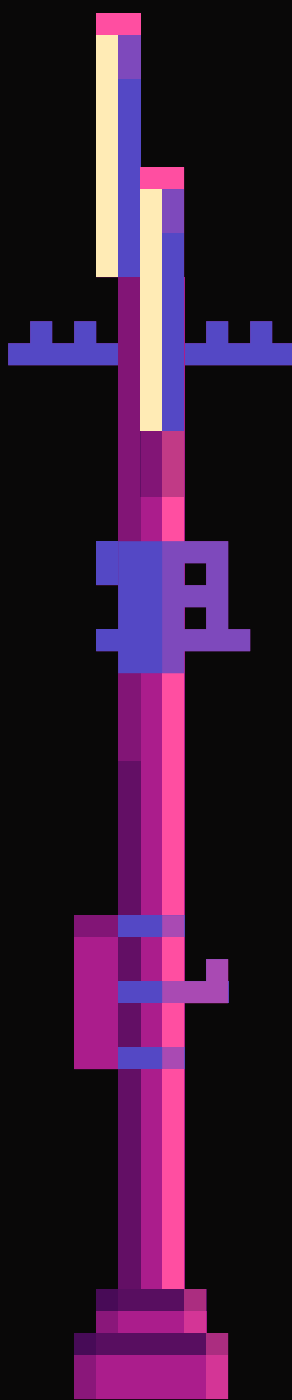




BUT.

But for objects which are quite different in brightness like, sun and a normal light bulb, assigning both of their brightness values between 0 and 1 seems impractical.

Here, comes the role of HDR. As, they can hold values for colours much grater than 1. So, bright things can really shine, dark areas stay dark, and we don't lose small details—even in very bright or dark scenes.

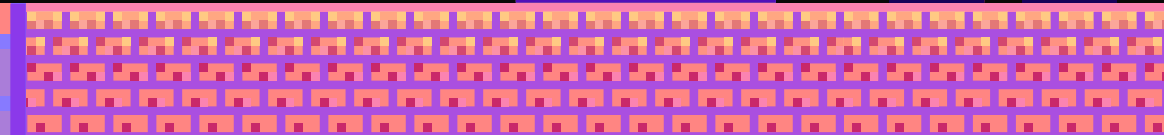
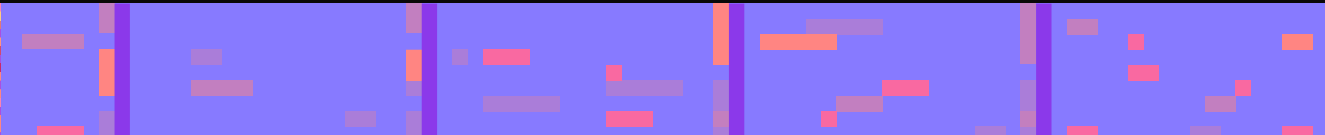
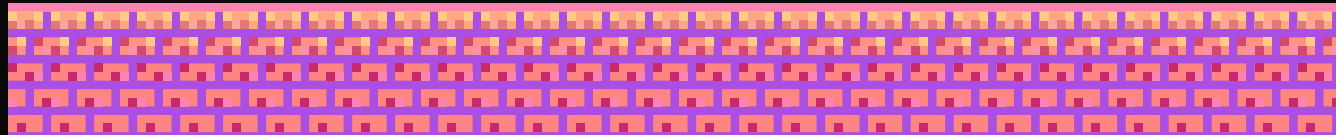
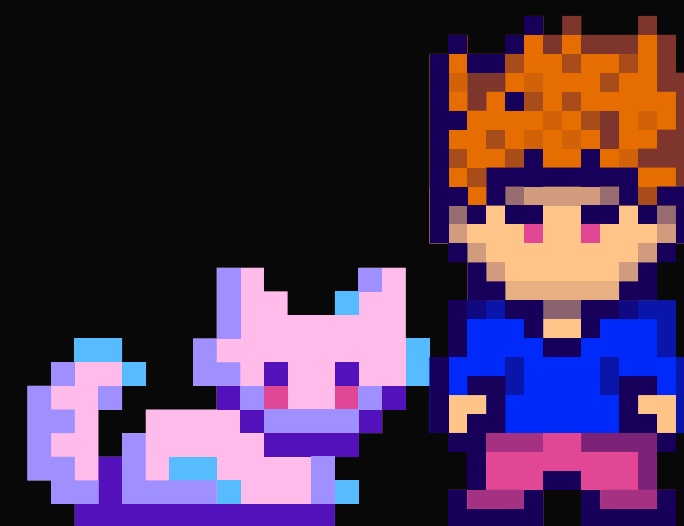




SOLVE: WHY SHOULD WE USE HDR ?

In real life, our eyes can see both: The bright sun ☀ And the shadows under a tree 🌳 at the same time.

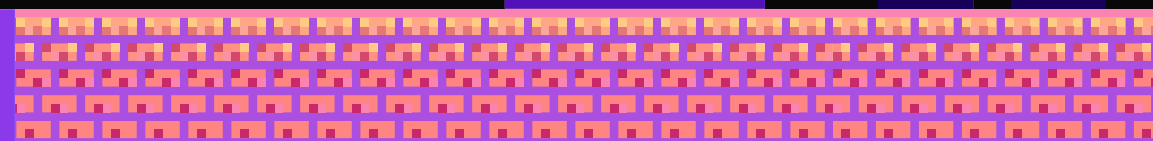
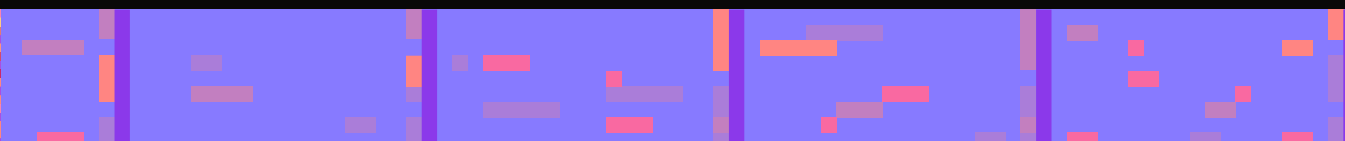
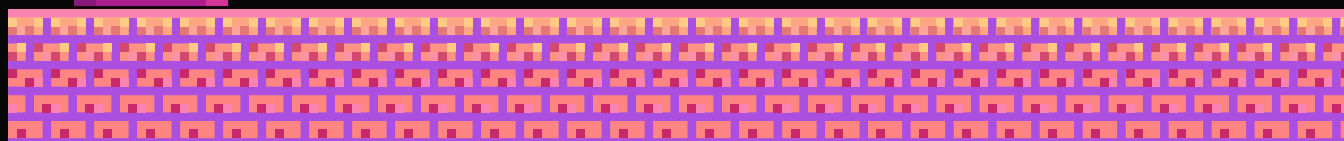
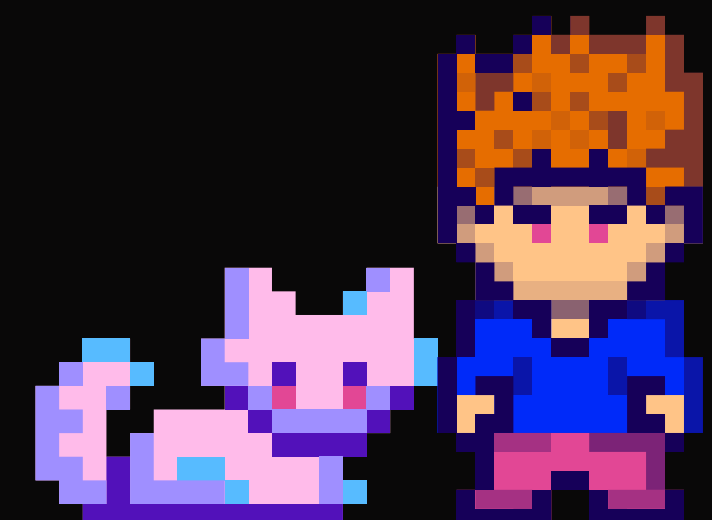
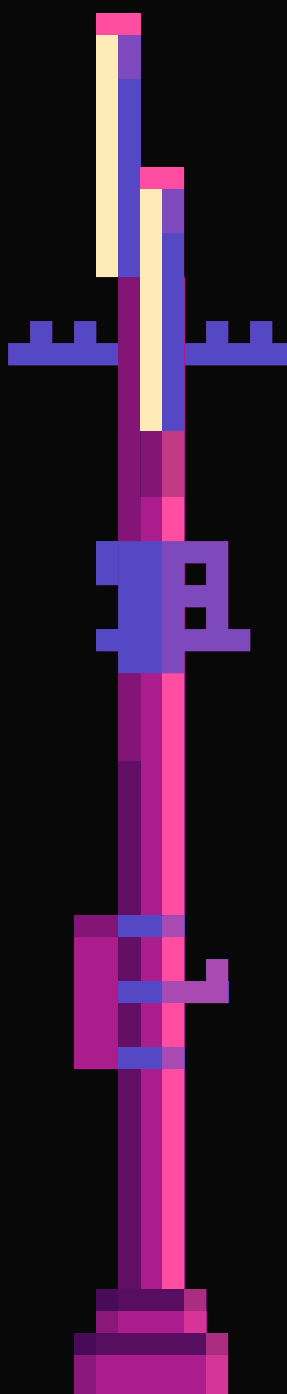
In graphics, standard rendering (LDR – Low Dynamic Range) often can't handle both well – shadows become too dark or lights get blown out. HDR fixes that by allowing us to give higher brightness levels to highly bright objects and similarly for low brightness object.

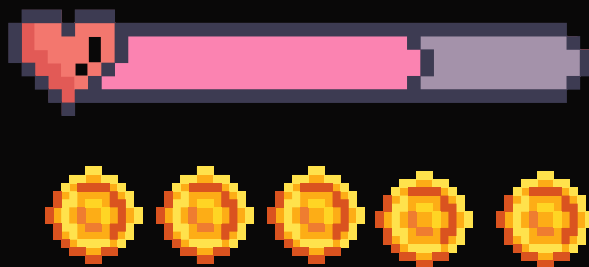




BUT WE CANNOT JUST USE ONLY HDR.

Because when we use HDR but show the result on an LDR screen, the bright colors (like sunlight) get cut off at 1.0 – this makes them look pure white and we lose details. It feels like we are not using HDR at all, because the display can't show values above 1.0. So, to keep the details and make the image look good on normal screens, we use a method called "Tone Mapping".



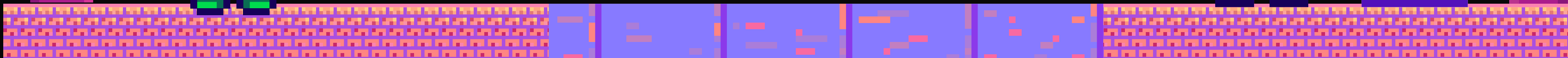
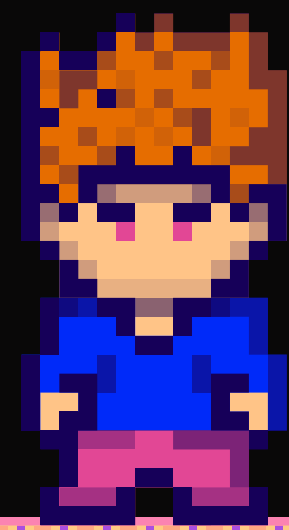
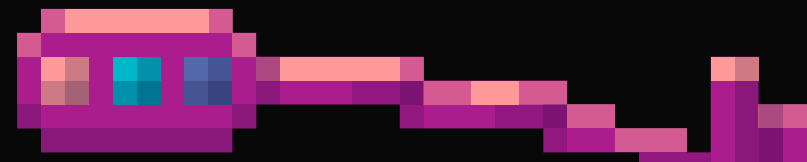


MENU

START



POST PROCESSING

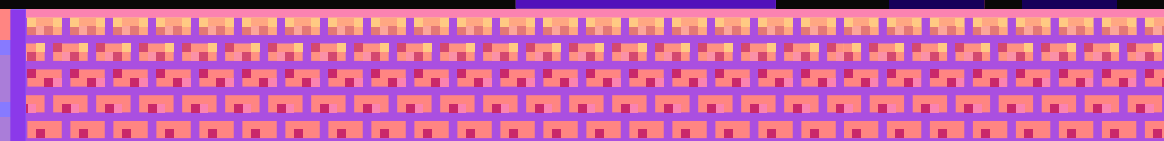
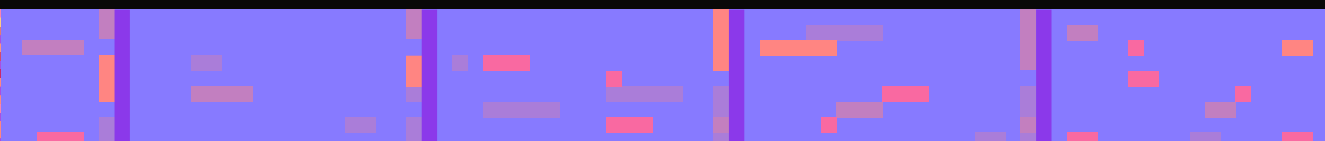
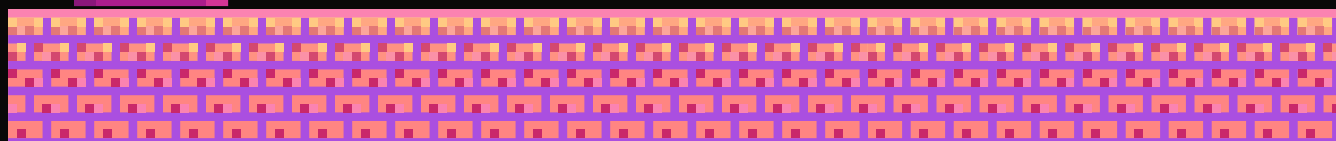
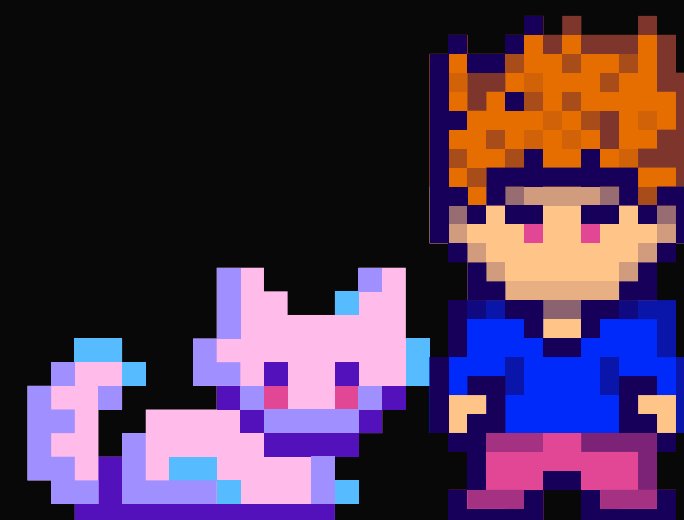
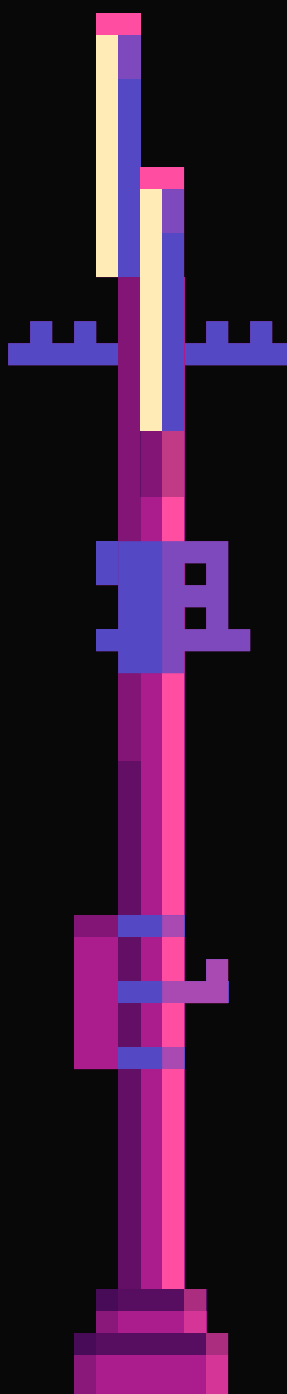




tone mapping

As already mentioned, Tone Mapping is a process that converts HDR colors (above 1.0) into LDR range (0.0 – 1.0)

It does this without losing important details like shadows or highlights. It allows us to keep both bright and dark parts of the scene visible and balanced.

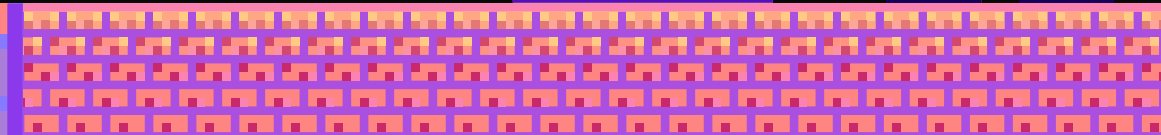
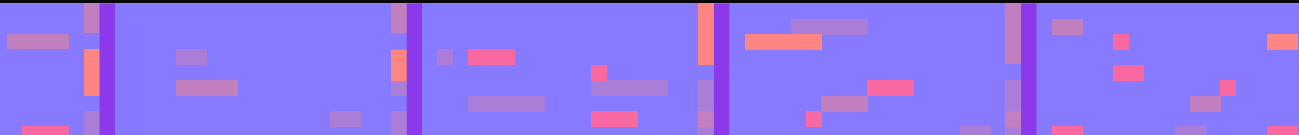
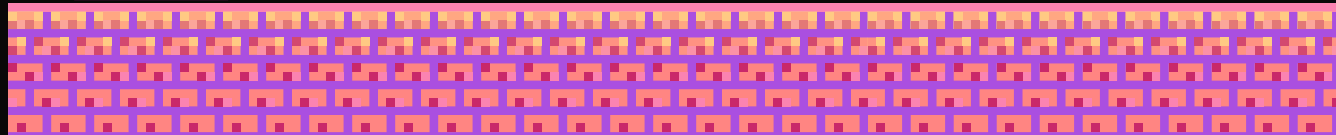
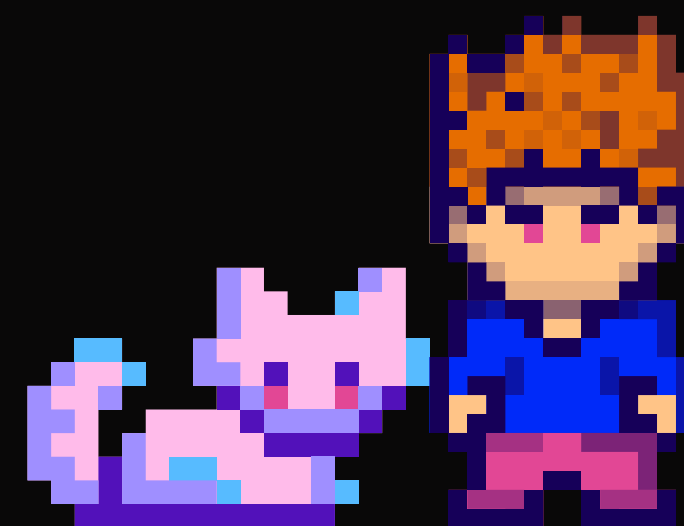




I. REINHARD TONE MAPPING

A basic tone mapping algorithm. It helps prevent over-bright areas from becoming plain white by evenly balancing brightness.

Often used with gamma correction to keep the image realistic. Good for simple scenes. Keeps most lighting details, but might make darker areas look less sharp.





Formula used in this algorithm:

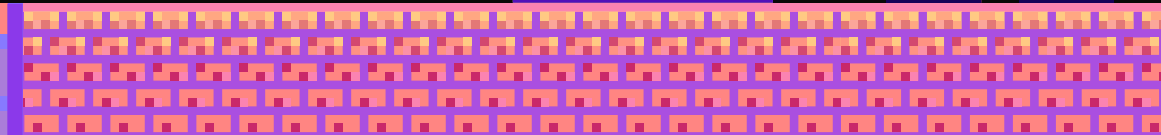
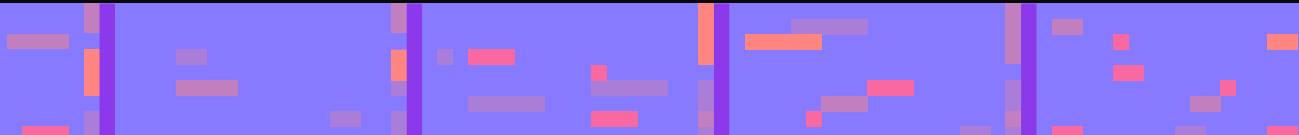
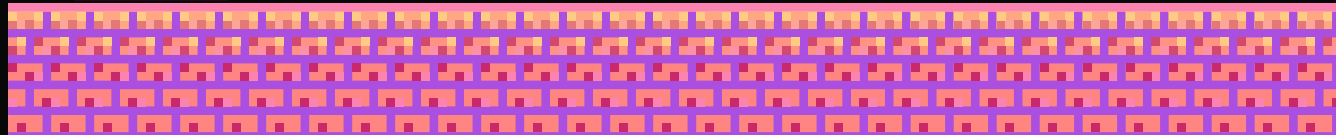
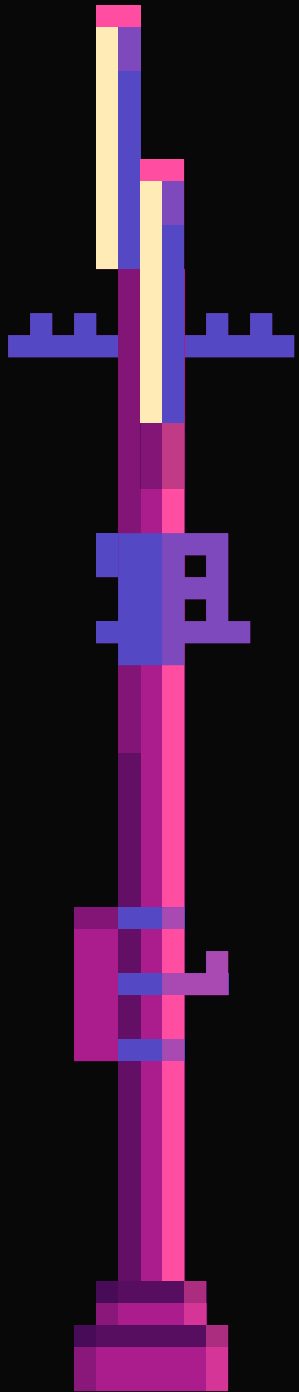
$$\text{mapped} = \left(\frac{\text{hdrColor}}{\text{hdrColor} + 1} \right)^{\frac{1}{\gamma}}$$

Here,

gamma is the correction factor.
(2.2 standard value)

hdrColor is the HDR input color to be
changed for the LDR screen.

mapped is the final colour after tone mapping + gamma correction.



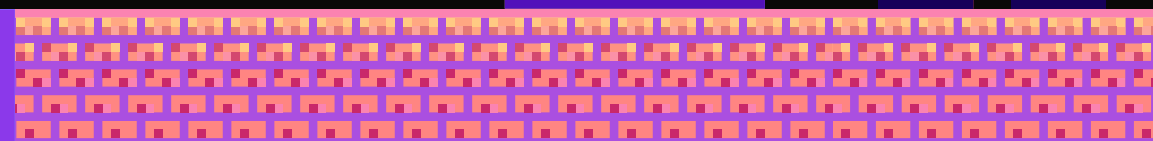
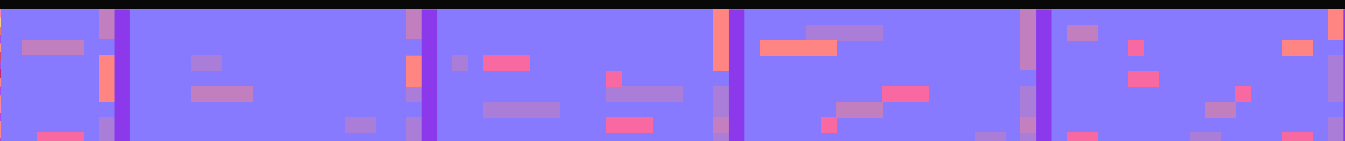
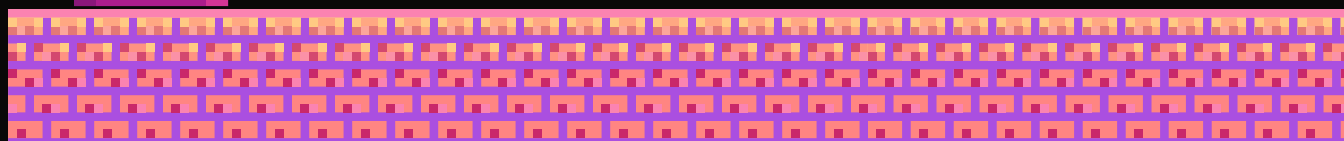
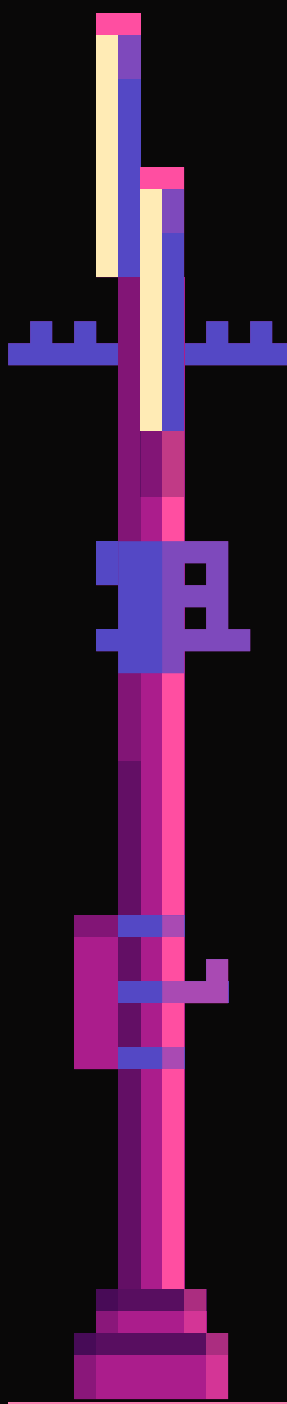


II. EXPOSURE ALGORITHM

It is seen that A higher exposure makes dark areas brighter, lower exposure keeps bright areas in control. Here this method is useful for scenes where lighting changes — lets us adjust brightness without losing detail.

for this purpose it, uses an exposure value constant to control scene brightness.

This method is great for day-night cycles.





MENU

START



THANK
YOU

