

الله اعلم
الحمد لله



دانشگاه شهید باهنر کرمان

دانشکده فنی و مهندسی

پروژه کارشناسی رشته مهندسی کامپیووتر

عنوان: پیاده سازی Real-time Render Engine

استاد راهنما: سرکار خانم دکتر قاسمیان

دانشجو: احمد منصوری بیدخوانی

تابستان 1400

چکیده

همزمان با پیدایش کامپیوتر ها، تلاش ها برای بهره بردن از توان آنها برای ساخت ابزار های تصویر کردن^۱ داده ها و ابزار های مربوطه برای استفاده از قدرت کامپیوتر در زمینه های نظامی، فیلم و انیمیشن، شبیه سازی و بازی سازی شروع شد.

طراحی و پیاده سازی موتور های رندر سه بعدی^۲ به صورت کلی دارای پایه ها و دانشی یکسان از نحوه کار پردازنده ها و ریاضیات است.

هدف این پروژه مطالعه و ساخت یک موتور رندر سه بعدی بی درنگ^۳ و بررسی تکنیک ها و روش های جایگزین برای استفاده در صنایع و پروژه های مختلف است. render engine ها پایه های اصلی تمامی شبیه ساز ها، ابزار های صنعتی، بازی های کامپیوتری و ابزارآلات مربوط به صنعت فیلم و انیمیشن می باشند که در دسته بندی های مختلفی ساخته می شوند، این پروژه مربوط به ساختن دسته بندی خاصی از این موتور ها یعنی دسته بندی real time می شود که کاربرد زیادی در تمامی زمینه های بالا به خصوص صنعت بازی های رایانه ای دارد. در انتهای این گزارش فصلی برای نمایش برخی از قابلیت های یک realtime render engine اختصاص داده شده.

visualize[†]
3D render engine[†]
real-time render engine[†]

فهرست مطالب

4

Renderer 1

5	پنجره ها و کانتکسٹ ها	1.1
5	openGL	1.1.1
6	GLFW	2.1.1
7	GLAD	3.1.1
8	تصویر کردن داده ها	2.1
8	Vertex Data	1.2.1
10	Shaders	2.2.1
14	Upload Data to GPU	3.2.1
17	Render Loop	4.2.1
21	Textures	5.2.1
25	دوربین و حرکت در جهان	3.1
25	Transformations	1.3.1
30	نورپردازی و سایه ها	4.1
30	Directional Lights	1.4.1
30	Lighting models	2.4.1
33	Shadows	3.4.1
37	کاربردها	5.1
37	Use Cases	1.5.1
39	3D viewer	2.5.1
40	Games	3.5.1

فهرست تصاویر

٩	glfw logo	1.1
١٠	Graphic pipeline: from learnopengl.com	2.1
١٥	vertex attribute linking	3.1
١٧	hello triangle	4.1
٢٠	colorful triangle	5.1
٢٤	textured triangle	6.1
٢٧	perspective frustum	7.1
٢٩	cubes in 3D	8.1
٣٢	blinn-phong shading	9.1
٣٢	a gun specular map	12.1
٣٣	blinn-phong shading	10.1
٣٤	phong shaded gun	11.1
٣٤	absence of light - shadow	13.1
٣٦	scene with shadow	14.1
٣٩	model in microsoft 3d viewer	15.1
٤٩	model in my project wit a green point light and ground	16.1
٥٠	original flappy bird	17.1
٥٢	my project flappy bird	18.1

فصل ۱

Renderer

مقدمه

real time rendering به تولید متناسب تصاویر بر مبنای تغییرات مدل‌ها یا دوربین در فضای نرم افزار گفته می‌شود. ابزار دستیابی به این هدف real-time rendering engine هاستند. هدف این پروژه مطالعه ساختارها و تکنیک‌ها در این زمینه و پیاده‌سازی آن‌ها به منظور دستیابی به این توانایی است.

تاریخچه استفاده از گرافیک کامپیوتری^۱ به دهه پنجاه قرن بیستم میلادی در عرصه جنگ سرد و بعدتر در زمینه رصد کردن تصویری را در ها برمی‌گردد، در دهه شصت، استفاده از این تکنولوژی در زمینه بازی‌های ویدیوئی^۲ منجر به تولید یکی از مهم ترین بازی‌های ویدیویی یعنی spacewar انجامید و سبب علاقمندی به این نوع سرگرمی شد، دهه هفتاد و با اوچ گیری پیشرفت‌های سخت افزاری، پیشرفت‌های چشمگیری در زمینه گرافیک کامپیوتر به وجود آمد که این روند تا سال‌های اخیر و با معرفی کارت گرافیک‌هایی با قابلیت استفاده بی‌درنگ از الگوریتم‌های دنبال کردن مسیر نور^۳ ادامه داشته است.

ترتیب نگارش بخش‌های مختلف پایان نامه بر اساس مراحل ساخت یک real-time render engine خواهد بود.

Computer Graphics^۱

video games^۲

RTX (real-time ray tracing)^۳

۱.۱ پنجره ها و کانتکست ها

openGL ۱.۱.۱

یک *OpenGL api* چند زبانه^۱ و چند سکویی^۲ است که برای به تصویر کشیدن تصاویر دو بعدی و سه بعدی با استفاده از بردارها^۳ اسفاده می شود، معمولاً از *OpenGL* برای برقراری ارتباط با واحد پردازش گرافیکی (*GPU*) و بهره بردن از سرعت سخت افزار مخصوص برای رندر استفاده می شود.

همچنین *api* های دیگری نیز برای استفاده از قدرت سخت افزاری و پردازنده گرافیکی وجود دارند، *Vulkan* نیز مانند *openGL* چند زبانی و چند سکویی است، *DirectX* به صورت انحصاری توسط مایکروسافت توسعه می یابد و در سیستم عامل ویندوز استفاده می شود ، شرکت *Apple* از *api* اختصاصی خود به نام *Metal* به صورت انحصاری پشتیبانی می کند، دلیل انتخاب *openGL* در این پروژه چندسکویی بودن و ساختار ساده تر برای پروژه های آموزشی در زمینه real-time rendering می باشد.

نباید *OpenGL* را با یک کتابخانه^۴ اشتباه گرفت، **OpenGL** به صورت یک interface و یک فرآداد انتزاعی در ورژن های مختلف ارائه می شود که فروشنده گان و سازنده گان (*vendor*) مختلف باید پیاده سازی ای منطبق با این فرآداد را انجام دهند. پس از نصب درایور مربوط به پردازنده گرافیکی، برنامه نویس قابلیت دسترسی به تابع های مختلف که توسط *vendor* پیاده سازی شده را خواهد داشت. برای استفاده از *OpenGL* نیاز به ابزار های دیگری نیز داریم، ابتدا نیاز داریم که یک *window* و یک *context* تعریف کنیم، برای این کار از **GLFW** استفاده می کنیم.

Cross Language^۱
Cross Platform^۲
vectors^۳
library^۴

GLFW ۲.۱.۱

یک کتابخانه برای ساختن *window* و *context* است، این کتابخانه به زبان C نوشته شده و *binding* های مختلف آن به زبان های مختلف موجود است، این کتابخانه همچنین توانایی کنترل کردن ورودی های مختلف مثل *keyboard*, *mouse*, *joystick* را دارد، ما برای استفاده از *OpenGL* یا *Window* یا *Context* نیاز به این کتابخانه یا مشابه آن داریم زیرا *OpenGL* هیچگونه قابلیت پیشفرضی برای مدیریت *Window* یا *Context* ندارد. همچنین **GLFW** یک کتابخانه چندسکویی است و می توانیم آن را در سیستم عامل های *Windows*, *Linux*, *Mac OS X* و *Android* استفاده کنیم، پروژه من نیز چندسکویی است، پس می توانیم از این کتابخانه سبک و چندسکویی استفاده کنیم. مثلاً برای کنترل *framebuffers* و *state* می توانیم از *OpenGL* استفاده کنیم، برای کنترل *Windows* می توانیم از **GLFW** استفاده کنیم، *GLFW* به وسیله امکانات فراهم شده در سطح سیستم عامل برای ما فراهم می کند، همچنین یک *Context* را می توانیم به عنوان یک شی در نظر بگیریم که تمامی اطلاعات *OpenGL* را به همراه دارد، اطلاعاتی مانند *framebuffers* و *state* را در *Windows* می توانیم در *OpenGL* استفاده کنیم و به صورت جداگانه نگهداری می شوند. برای کنترل *Windows* می توانیم از *Windows API* استفاده کنیم، برای کنترل *framebuffers* و *state* می توانیم از *OpenGL* استفاده کنیم، برای کنترل *mouse* می توانیم از *GLFW* استفاده کنیم، برای کنترل *keyboards* می توانیم از *GLFW* استفاده کنیم و به صورت مستقیم ورودی را دریافت کنیم.

حالا که به *Window* و *Context* دسترسی داریم، باید دسترسی به تابع های *OpenGL* فراهم کنیم، برای این کار از **GLAD** استفاده می کنیم.



شکل ۱.۱ : **glfw logo**

GLAD ۳.۱.۱

کتابخانه ای برای *load* کردن نشانگر ها^۱ ها به توابع *opengl* در زمان اجرای برنامه^۲ است. این کتابخانه یکی از کتابخانه های OpenGL Loading Library است، برای کار با *opengl* حتما باید یکی از این کتابخانه هارا مورد استفاده قرار دهیم تا بتوانیم به توابع *opengl* دسترسی داشته باشیم، این کتابخانه ها هم ویژگی های هسته^۳ که توسط *opengl* مشخص شده و هم ویژگی های افزونه^۴ که توسط Vendor ها به پیاده سازی آن ها از *opengl* اضافه شده، علاوه بر این دیگر نیازی به اضافه کردن فایل های مربوط به *opengl* نیست و این فایل ها به صورت خودکار همه موارد را تنظیم می کنند. Glad یک generator است که براساس پارامتر هایی که کاربر انتخاب می کند یک فایل حاوی تمامی تعریف های مربوط به *constant* و *function* ها و ... به ما ارائه می کند، بعد از دانلود این فایل و اضافه کردن به آن به پروژه از طریق کد زیر می توانیم تمامی نشانگرها به توابع *OpenGL* را در *runtime* بارگزاری کنیم.

برنامه ۱.۱ *load opengl function pointer*

```
1 // glad: load all OpenGL function pointers
2 // -----
3 if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
4 {
5     std::cout << "Failed to initialize GLAD" << std::endl;
6 }
```

پس از ساختن پنجره و *context* و بارگزاری توابع، حالا آماده استفاده از *openGL* هستیم.

pointers^۱
runtime^۲
Core^۳
extension^۴

۲.۱ تصویر کردن داده ها

Vertex Data ۱.۲.۱

برای *render* کردن تصاویر نیاز به اطلاعاتی داریم، با مثلث شروع می کنیم، مثلث در گرافیک کامپیوتری جایگاه ویژه ای دارد، مثلث ساده ترین شکلی است که تشکیل سطح میدهد، برای رسم کردن یک مثلث در صفحه نیاز به سه نقطه داریم، با متصل کردن این سه نقطه به یکدیگر مثلث ساخته می شود، برای رسم مثلث در *opengl* نیز شرایط به همین شکل است، ما نیاز به سه نقطه داریم، تفاوت این نقاط با نقطه های روی صفحه در ابعاد آن است، تقاطع روی صفحه دو بعدی بودند، *opengl* نقاط را به صورت سه بعدی دریافت می کند، هر کدام از این نقاط متشكل از سه مقدار برای x, y, z هستند، این نقاط را می توان به صورت بردارهایی در *Normalized Device Coordinates* نمایش داد، یک بردار در NDC را به شکل رو به رو نمایش می دهیم:

$$\vec{P} = (x, y, z)$$

مقادیر z, y, x در این مختصات باید بین $[+1, -1]$ باشند، اگر مقداری خارج از این بازه باشد بر روی صفحه قابل مشاهده نیست. هر کدام از این نقاط را یک *Vertex* می نامیم.

بسته به درخواستی که از *opengl* می کنیم، نحوه برخورد با این نقاط و در نتیجه نحوه به تصویر کشیدن این نقاط روی صفحه تغییر می کند، به عنوان مثال می توانیم با این نقاط به شکل مثلث، نقطه یا خط و اشکال دیگری برخورد کنیم. برای برقراری ارتباط با *opengl* از زبان برنامه نویسی *C* استفاده می کنیم، برای رسم کردن مثلث در این مرحله آرایه زیر را تعریف می کنیم:

برنامه ۲.۱ points for a triangle

```
1 float vertices[] = {  
2     // x , y, z  
3     -0.5f, -0.5f, 0.0f, //p1  
4     0.5f, -0.5f, 0.0f, //p2  
5     0.0f, 0.5f, 0.0f //p3  
6 };
```

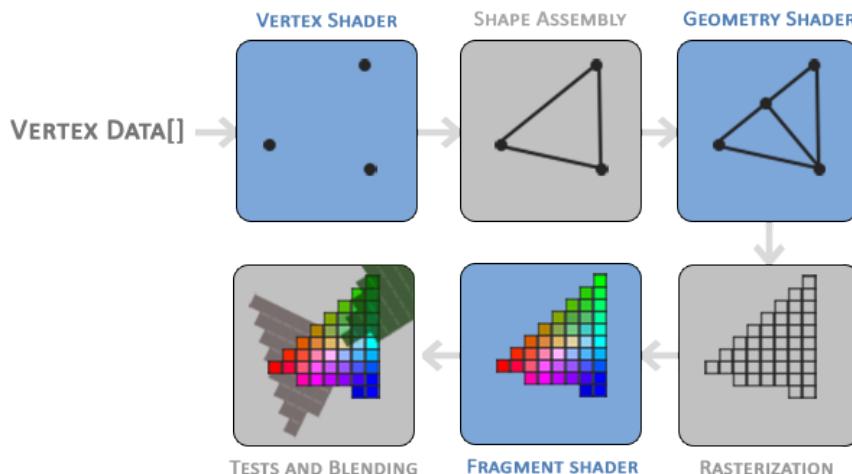
در قطعه کد بالا از ۹ عدد که همگی در بازه مشخص شده برای NDC هستند استفاده کردیم، توجه کنید که اعداد در یک آرایه و یه صورت پشت سر هم به برنامه داده شده اند، هیچگونه جداسازی یا طبقه بندی بر اساس نقاط مختلف صورت نگرفته و همچنین این مقادیر هنوز بر روی Gpu آپلود نشده اند، دسته بندی کردن این اطلاعات خام و آپلود بر روی Gpu در دو فصل بعد شرح داده خواهد شد. آرایه ای که تعریف کردیم تنها شامل مختصات نقاط مثلت بود، ما می توانیم هر گونه اطلاعاتی را به همین صورت در این آرایه اضافه کنیم و دسته بندی آن ها را مشخص کنیم و از آنها استفاده کنیم، برای مثال می توانیم اطلاعات مربوط به ... Color, Normal, Texture Coordinate را اضافه کنیم.

برای آپلود کردن داده ها بر روی پردازنده گرافیکی راه های مختلفی بسته به نیاز های مختلف وجود دارد، در بخش های بعدی چند نوع از این روش هارا می بینیم، برای تعریف کردن اطلاعاتی که باید بر روی Gpu آپلود شود باید آن ها را در برنامه های به نام Shader ها مشخص کنیم، در بخش بعدی درباره این برنامه ها صحبت می کنیم.

Shaders ۲.۲.۱

کارت های گرافیک امروزی، تشكیل شده از تعداد بسیار زیادی از هسته های پردازشی هستند که وظیفه اجرای برنامه های کوچکی به نام *Shader* ها را بر عهده دارند.

ها بیانگر Graphic Pipeline بر روی کارت های گرافیک هستند، این ابزار به ما قابلیت کنترل و کدنویسی هر کدام از این مراحل را می دهد، بر روی کارت گرافیک های امروزی تمامی shader ها به جز دو نوع از آن ها به صورت پیشفرض وجود دارد، این دو Vertex shader و Fragment shader هستند که حتما باید توسط برنامه نویس به کارت گرافیک داده شوند.



Graphic pipeline: from learnopengl.com شکل ۲.۱

به طور کلی وظیفه *Vertex Shader* انتقال یک نقطه از فضای NDC به فضای دیگر است، غالباً این فضا همان جهان بازی یا برنامه سه بعدی است، برای Transform کردن فضای بازی به فضایی دیگر از ماتریس ها استفاده می شود، به طوری که هر فضای دارای یک Transformation Matrix است، برای رسم کردن مثلث مانیزی به تغییر فضا نداریم و در NDC ادامه می دهیم. یک Vertex shader ساده به صورت زیر است:

برنامه ۳.۱: basic vertex shader

```
1 #version 330 core
2 layout (location = 0) in vec3 aPos;
```

```

4 void main()
5 {
6     gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
7 }

```

کد بالا ساده ترین مدل برای استفاده از vertex shader است، این کد به زبان GLSL(OpenGL Shading Language) نوشته شده است، در خط اول مانع ویژگی ها را که بیشتر توضیح داده بودیم را مشخص کرده ایم، خط ۲ یک متغیر از نوع vec3 به اسم aPos تعریف کرده ایم، با استفاده از *layout (location = 0)* در این خط مشخص کرده ایم که این متغیر در حافظه در چه موقعیتی قرار میگیرد، این ویژگی برای وقتی که بخواهیم مقادیر دیگری به جز مختصات نقطه، در یک آرایه ذخیره کنیم و به کارت گرافیک دهیم کارایی دارد. در نهایت متغیر (gl_Position) که نشان دهنده مکان این نقطه که در حال پردازش است می باشد را به وسیله مقادیری که از کد C خواندیم و مقدار چهارم که ۱ مقداردهی کردیم، مقدار چهارم در این فضا کاربردی ندارد اما در فضای سه بعدی و در perspective view به کار ما می آید.

مرحله بعد ساختن یک Fragment shader است، وظیفه این بخش از pipeline انجام محاسبات و تعیین رنگ پیکسل می باشد، تمامی محاسبات مربوط به نور، سایه، بازتاب و افکت های گرافیکی غالبا در این مرحله انجام می شود، البته این مقادیر در مراحل بعد ممکن است تغییر کنند. یک fragment shader ساده به صورت زیر است:

برنامه ۴.۱ basic fragment shader

```
1 #version 330 core
2 out vec4 FragColor;
3
4 void main()
5 {
6     FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
7 }
```

کد بالا مقدار خروجی برای رنگ این fragment را برابر با مقداری ثابت قرار می دهد، نوع متغیر FragColor از نوع vec4 تعریف شده و مقادیری که دریافت کرده به ترتیب معنی red, green, blue, alpha را می دهند، مقادیر باشد بین صفر و یک باشند.

حالا هر کدام از کدهای بالارا compile می کنیم و سپس به برنامه اصلی که روی Gpu قرار می گیرد متصل می کنیم، این برنامه را shader program می نامیم، قطعه کد پایین مراحل compile و link کردن shader program را نشان می دهد.

برنامه ۵.۱ compile and link shaders to shader program

```
1 // hold Vertex shader ID
2 unsigned int vertexShader;
3 // create a shader of vertex type
4 vertexShader = glCreateShader(GL_VERTEX_SHADER);
5 // upload source code
6 glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
7 // compile vertex shader source
8 glCompileShader(vertexShader);
9 // -----
10 // hold Fragment shader ID
11 unsigned int fragmentShader;
12 //create a shader of fragment type
13 fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
14 // upload source
15 glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
```

```

16 // compile fragment shader source
17 glCompileShader(fragmentShader);
18 // -----
19 // hold shader program ID
20 unsigned int shaderProgram;
21 // create a shader program
22 shaderProgram = glCreateProgram();
23 // attach vertex shader to program
24 glAttachShader(shaderProgram, vertexShader);
25 // attach fragment shader to program
26 glAttachShader(shaderProgram, fragmentShader);
27 glLinkProgram(shaderProgram); // link the program

```

حالا می توانیم از این shader program برای تصویر کردن داده ها استفاده کنیم، در بخش بعد داده ها را بر روی Gpu بارگزاری می کنیم.

Upload Data to GPU ۳.۲.۱

برای استفاده از مشخصات نقاطی که تعریف کردیم باید آن هارا روی memory کارت گرافیک آپلود کنیم، انتقال اطلاعات بین cpu و gpu نسبتاً کند است، پس سعی می کنیم اطلاعات هر چه بیشتری را در یک بار انتقال منتقل کنیم، برای مدیریت حافظه بر روی کارت گرافیک از vbo(vertex buffer object) استفاده می کنیم، این بافر ها قابلیت نگهداری تعداد زیادی از داده ها را داند، برای ساختن یک buffer در opengl و نگهداری مشخصه بافر ایجاد شده به صورت زیر عمل می کنیم:

creating a buffer object : برنامه ۶.۱

```
1 unsigned int VBO;  
2 glGenBuffers(1, &VBO);
```

برای استفاده کردن از این بافر و ارسال اطلاعات باید آن را bind کنیم:

binding to target gl_array_buffer : برنامه ۷.۱

```
1 glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

حالا که بافر bind شده است، می توانیم داده هایی که در آرایه vertices تعریف کردیم را به gpu memory منتقل کنیم:

uploading data for static draw : برنامه ۸.۱

```
1 glBufferData(GL_ARRAY_BUFFER,  
2             sizeof(vertices),  
3             vertices,  
4             GL_STATIC_DRAW);
```

حالا داده های ما بر روی gpu قرار گرفته اند، آرگومان آخری که به تابع بالا دادیم به این معنی است که این داده ها مستعد تغییر نیستند، یعنی نوشتن بر روی آن ها زیاد صورت نمی گیرد اما باید برای خوانده شدن به سرعت در دسترس باشند زیرا به مراتب خوانده می شوند، این به گرافیک کمک می کند تا داده هارا در جایی از حافظه قرار دهد تا این ویژگی ها را داشته باشد.

داده هایی که بر روی کارت گرافیک دادیم داده خام هستند، باید برای vertex shader مشخص کنیم که به چه

صورت باید داده هارا تفسیر کند، برای استفاده از ویژگی vertex shader ها در attribute باید نوع تفسیر داده هارا نیز به صورت دستی مشخص کنیم، به اینکار Linking Vertex Attribute می‌گویند.

```
float vertices = [
    a vertex - 0.5, -0.5, 0.0,
    a vertex 0.5, -0.5, 0.0,
    a vertex 0.0, 0.5, 0.0
]
```

شكل ۳.۱ vertex attribute linking

برای انجام اینکار باید مقدار چند ویژگی را بدانیم:

۱. اولین آرگومان برابر با مقداری است که برای متغیر aPos در vertex shader به وسیله layout (location) مخصوص کردیم که در این مورد برابر با صفر است.

۲. آرگومان دوم مشخص کننده تعداد متغیر های است که باید به عنوان یک vertex شناسایی شوند، این مقدار برای مثال ما برابر ۳ است.

۳. این آرگومان مشخص کننده نوع داده هایی است که بارگزاری شده، در این مورد float است.

۴. مشخص می کند که داده ها نیاز به نرمال سازی دارند یا خیر.

۵. این مقدار مشخص می کند چند byte داده باید برای این vertex خوانده شود، به این مقدار stride می‌گویند.

۶. مقدار آخر در مورد مثال ما کاربرد ندارد، در مثال های بعدی می بینیم که مقادیر مربوط به رنگ و دیگر ویژگی های یک نقطه را در به صورت پیوسته در آرایه vertices اضافه می کنیم، آنگاه باید از offset برای مشخص کردن هر کدام از این ویژگی ها استفاده کنیم.

شكل ۳.۱ متناسب با توضیحات ساخته شده.

link vertex attribute to vertex data : ۹.۱ برنامه

```

1 glVertexAttribPointer(
2     0, // layout (location = 0)
3     3, // 3 value for this vertex exists
4     GL_FLOAT, // type of each value in vertex
5     GL_FALSE, // no need to normalize data
6     3 * sizeof(float), // 3 (size of) float in each vertex
7     (void*)0 //no offset
8 );
9
10 glEnableVertexAttribArray(0);

```

در پروژه های بزرگ تر انجام دادن این عملیات برای تک تک اشیاء موجود در بازی بیهوده و زمان گیر است، برای همین از یکی دیگر از انواع بافر ها به اسم vertex array object استفاده می کنیم، این بافر تمامی مراحل قبل و فعال سازی vertex attribute ها را ذخیره می کند و دفعات بعد نیازی به انجام تمامی این مراحل نیست و ما فقط باید bind را VAO کنیم:

link vertex attribute to vertex data : ۱۰.۱ برنامه

```

1 unsigned int VAO;
2 glGenVertexArrays(1, &VAO);
3 // 2. copy our vertices array in a buffer for OpenGL to use
4 glBindBuffer(GL_ARRAY_BUFFER, VBO);
5 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
6               GL_STATIC_DRAW);
7 // 3. then set our vertex attributes pointers
8 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (
9     void*)0);
10 glEnableVertexAttribArray(0);

```

حالا می توانیم با استفاده vao و shader program مثلث را بر روی صفحه رسم کنیم، این کار را در بخش بعدی

انجام می دهیم.

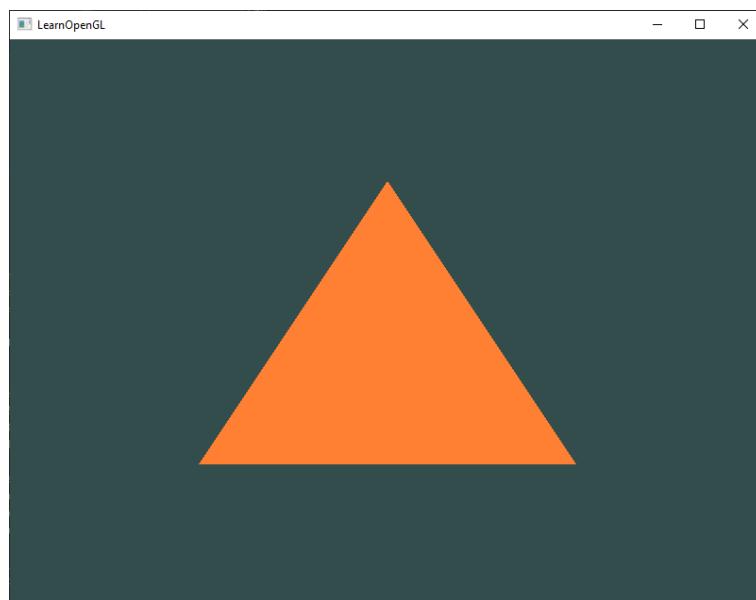
Render Loop ۴.۲.۱

حالا برای رسم کردن مثلث نیاز به کدهای زیر داریم:

link vertex attribute to vertex data : ۱۱.۱

```
1 glUseProgram(shaderProgram);  
2 glBindVertexArray(VAO);  
3 glDrawArrays(GL_TRIANGLES, 0, 3);
```

ابتدا shader program را فعال می کنیم، سپس VAO که ساختیم را bind می کنیم، حالا تمامی داده ها و تفسیر ها آماده هستند، برای رسم از تابع خط آخر می خواهیم که با داده ها تشکیل مثلث بدهد، ابتدای و انتهای بایت هایی که باید از vertex array بخواند را مشخص می کنیم، پس کامپایل کردن و اجرای برنامه با شکل زیر رو به رو می شویم.



شکل ۴.۱: hello triangle

برای اینکه بتوانیم دیگر ویژگی های مدل هارا به Gpu ارسال کنیم و از آن ها استفاده کنیم می توانیم از چند attribute یا از نوعی از متغیر ها به اسم uniform استفاده کنیم، در ابتدا برای افروختن رنگ به هر کدام از vertex ها vertex shader استفاده می کنیم، برای این کار کد vertex shader را به شکل زیر تغییر می دهیم:

برنامه ۱۲.۱ declare and use aColor :

```

1 #version 330 core
2 layout (location = 0) in vec3 aPos;
3 layout (location = 1) in vec3 aColor
4
5 out vec3 ourColor;
6
7 void main()
8 {
9     gl_Position = vec4(aPos, 1.0);
10    ourColor = aColor
11 }
```

و کد fragment shader را به شکل زیر می نویسیم تا بتوانیم از مقادیر رنگ استفاده کنیم:

برنامه ۱۳.۱ render fragment with color from vertex shader :

```

1 #version 330 core
2 out vec4 FragColor;
3 in vec3 ourColor;
4
5 void main()
6 {
7     FragColor = vec4(ourColor, 1.0);
8 }
```

حالا داده های مربوط به رنگ هر vertex را در انتهای داده های مربوط به همان vertex اضافه می کنیم، کد به شکل زیر تغییر می کند:

برنامه ۱۴.۱ □ *position and color data in one array*

```
1 float vertices[] = {  
2     // positions           // colors  
3     0.5f, -0.5f, 0.0f,   1.0f, 0.0f, 0.0f,    // bottom right  
4     -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,    // bottom left  
5     0.0f,  0.5f, 0.0f,   0.0f, 0.0f, 1.0f      // top  
6 };
```

داده هارا به شکل قبل به VBO اضافه می کنیم، سپس bind VAO را می کنیم، یک مرحله جدید در link برای رنگ ها اضافه شده است، این مرحله به شکل زیر است:

برنامه ۱۵.۱ □ *position and color data in one array*

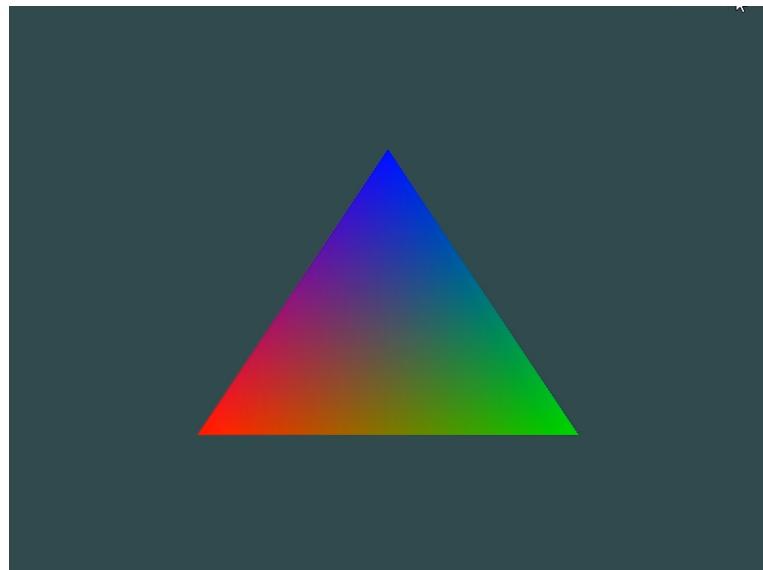
```
1 // position attribute  
2 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);  
3 glEnableVertexAttribArray(0);  
4 // color attribute  
5 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));  
6 glEnableVertexAttribArray(1);
```

همانطور که می بینیم در این مرحله متغیر های مربوط offset و stride به شکل جدید برای استفاده از و تفسیر تمام داده برای gpu تغییر کرده اند، در خط ۵ کد بالا متغیر offset نشان دهنده این است که برای خواندن سه مقدار برای رنگ، باید offset را برابر ۳ قرار دهیم تا سه مقدار اول که برای داده های مکان نقطه تعریف کرده ایم در نظر نگیرد، همچنین متغیر stride را برابر با شش قرار دادیم، این به معنی این است که هر ۶ مقدار در آرایه بیانگر ویژگی ها برای یک نقطه متفاوت است.

رنگ مثلث به صورت زیر تغییر می کند:

در مثال بالا OpenGL مقدار رنگ سه نقطه را از ما دریافت کرد و به صورت خودکار مقدار رنگ بین این نقاط را به صورت خطی interpolate کرد.

این روش برای مدل ها و اشیاء سه بعدی نه تنها کاربردی نیست بلکه هزینه زیادی دارد، روش بهتر برای مقدار دهی



شکل ۱: colorful triangle

به رنگ و ایجاد جزئیات برای اشیاء استفاده از texture ها است، در بخش بعد درباره تکسچرها توضیح می دهم.

Textures ۵.۲.۱

تصور کنید می خواهیم یک دیوار را شبیه سازی کنیم، دیوار را می توانیم با چهار نقطه با استفاده از EBO(element buffer object) بسازیم، این چهار نقطه برای ما تشکیل یک مریع یا مستطیل(دو مثلث که از وتر بر روی یکدیگر فرار گرفته اند) بسته به موقعیت نقاط می دهنند. مرحله بعدی اضافه کردن جزئیات رنگ یعنی شکل آجر ها بر روی دیوار است، این کار به سادگی امکان پذیر نیست، زیرا ما فقط قابلیت تعیین چهار رنگ برای چهار نقطه را داریم، پس نمی توانیم با چهار رنگ وengl linear interpolation که بین این رنگ ها انجام می دهد یک دیوار آجری رندر کنیم، راه حل این مشکل استفاده از یک texture است، به صورت ساده می توانیم عکس یک دیوار آجری را بهengl بدهیم و بخواهیم که به ازای هر fragment از یکی از pixel های عکسی که ما به عنوان texture آپلود کرده ایم استفاده کند. برای استفاده از یک texture ما باید مراحل زیر را انجام دهیم.

.۱. اضافه کردن texture coordinates به آرایه داده ها و آپلود آن بر روی .vertex attribute

.۲. خواندن عکس از حافظه و آپلود گردن آن بر روی .Gpu

.۳. تنظیم کردن حداقل برخی از ویژگی های تکسچر که به صورت پیشفرض مقداری ندارند.

.۴. استفاده از تابع sample برای GLSL کردن texture

در مرحله اول مقادیر texture coordinate را به آرایه داده ها اضافه می کنیم:

برنامه ۱۶.۱: *position and color data in one array*

```
1 float vertices[] = {  
2     // positions           // colors           // texture coords  
3     0.5f,  0.5f,  0.0f,   1.0f,  0.0f,  0.0f,   1.0f,  1.0f,  
4     0.5f, -0.5f,  0.0f,   0.0f,  1.0f,  0.0f,   1.0f,  0.0f,  
5    -0.5f, -0.5f,  0.0f,   0.0f,  0.0f,  1.0f,   0.0f,  0.0f,  
6    -0.5f,  0.5f,  0.0f,   1.0f,  1.0f,  0.0f,   0.0f,  1.0f  
7};
```

دقت کنید که باید مراحل linking vertex attribute را برای مقادیر جدید انجام دهیم.
حالا عکس را از دیسک می خوانیم، برای این کار از **stbi_image** استفاده می کنیم، این کتابخانه به صورت single header library در دسترس است، این گونه کتابخانه در یک فایل نوشته شده اند و استفاده درست از آن ها با استفاده از **define** است.

پس از باز کردن عکس در کد باید یک texture بسازیم، ابتدا یک شی texture می سازیم و سپس می توانیم داده های عکسی که خواندیم را بر روی تکسچری که ساختیم upload کنیم، اینکار به شیوه زیر انجام می گیرد:

برنامه ۱۷.۱: *load and upload data to gpu*

```
1 unsigned int texture; // hold texture ID  
2 glGenTextures(1, &texture); // create texture object on gpu  
3 // bind texture to 2d_texture target  
4 glBindTexture(GL_TEXTURE_2D, texture);  
5 // upload image data to texture that is bound i.e texture variable  
6 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,  
7             0, GL_RGB, GL_UNSIGNED_BYTE, data); // data is the Image  
8 // generating mip maps for this texture  
9 glGenerateMipmap(GL_TEXTURE_2D);
```

نیاز داریم که برخی مقادیر را برای texture تنظیم کنیم و گرنه با یک تکسچر سیاه رو به رو می شویم، مقادیر زیر برای تنظیم تکرار شدن تصویر در جهات مختلف و مشخص کردن الگوریتم مناسب برای زمان کوچک نمایی با بزرگ نمایی تصویر است، به صورت زیر این مقادیر را تنظیم می کنیم:

برنامه ۱۸.۱ \square
set minimum setting for a texture

```
1 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);  
2 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);  
3 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  
                 GL_LINEAR_MIPMAP_LINEAR);  
4 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

مختصات تکسچر را در vertex shader به صورت یک attribute جدید تعریف می کنیم و به مرحله in، ارسال این داده ها بین مراحل مختلف pipeline را با استفاده از متغیر هایی که با fragment shader می فرستیم، ارسال این داده ها بین مراحل مختلف تعیین می شود(در این مورد out تعیین می شوند انجام می دهیم، به این صورت که متغیر را در مرحله ای که زودتر انجام می شود) در این مورد (vertex shader) به صورت out تعیین می کنیم و در مرحله بعد آن را با in تعیین می کنیم، دقت کنید باید نام متغیر دقیقاً برابر باشد، کد زیر به vertex shader اضافه می کنیم:

برنامه ۱۹.۱ \square
vertex shader to use texture

```
1 layout (location = 2) in vec2 aTexCoord;  
2 out vec2 TexCoord; // vec2 because of 2d image  
3 ...  
4 void main()  
5 {  
6     ...  
7     TexCoord = aTexCoord;  
8 }
```

در قسمت fragment shader نیز کد به شکل زیر تغییر می کند:

برنامه ۲۰.۱ \square
fragment shader to use texture

```
1 #version 330 core  
2 out vec4 FragColor;  
3 ...  
4 in vec2 TexCoord;  
5 ...  
6 uniform sampler2D ourTexture;
```

```
7
8 void main()
9 {
10    //built in texture function for sampling texture to fragment
11    FragColor = texture(ourTexture, TexCoord);
12 }
```

نتیجه به شکل زیر در می آید:



شکل ۶.۱ textured triangle

۳.۱ دورین و حرکت در جهان

Transformations ۱.۳.۱

می توانیم با دو مثلث یک مربع درست کنیم، و با شش مربع یک مکعب تشکیل دهیم، و با استفاده از یک حلقه و چند draw call چند مکعب بسازیم، اما در حال حاضر این کار فایده ای ندارد چون تمامی مکعب ها بر روی یکدیگر قرار می گیرند و ما فقط یک مکعب را به صورت دو بعدی می بینیم، برای دیدن یک مکعب به یک محیط که شیوه ساز سه بعد باشد نیاز داریم، می توانیم مختصات مکعب هارا در حلقه تغییر دهیم و در قسمت های مختلف صفحه رندر کنیم، به این کار translate کردن می گوییم، این عمل را با استفاده از یک ماتریس 4×4 انجام می دهیم، یک matrix به شکل زیر است:

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

مقادیر (x, y, z) به ترتیب نشان دهنده تغییرات بر بردار متناسب با خود هستند، می توانیم هر کدام از نقاط مدل خود را توسط ماتریس بالا به نقطه جدید خود منتقل کنیم، ماتریس های rotation و scale نیز به ترتیب برای چرخش حول محور های مشخص و تغییر مقیاس استفاده می شوند، به حاصل ضرب این سه جزء یک model matrix می گویند.

پس از ضرب کردن model در مختصات هر نقطه از شکل خود، اصطلاحاً مختصات نقاط را در local space به world space بردیم.

برای انجام عملیات ریاضی مربوط به opengl بر روی cpu از کتابخانه **GLM** استفاده می‌کنیم، این کتابخانه دارای کلاس‌ها و توابعی است که نمایانگر ساختارهای ریاضیاتی مانند vector و matrix ها است، همچنین تعداد دیگری از توابع را دارد که برای ساده سازی کار ما فراهم شده‌اند، یکی از این توابع `glm::LookAt` است، می‌توانیم از این کلاس برای تشکیل یک `Camera` استفاده کنیم که نمایانگر `view matrix` است.

Camera

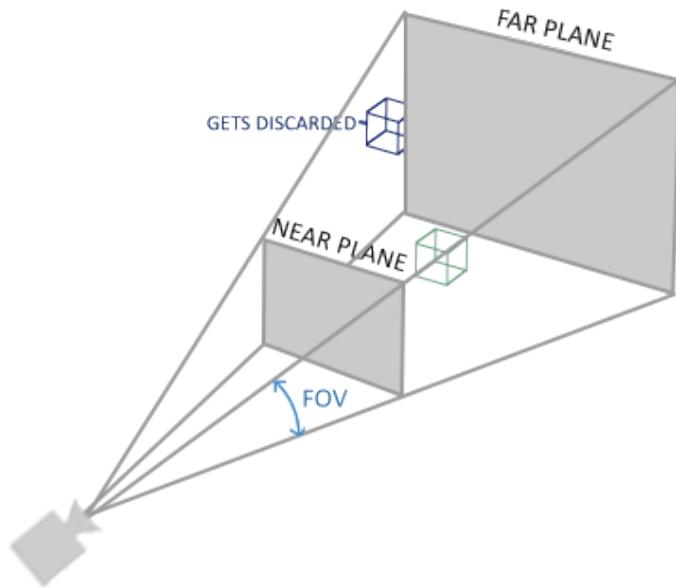
در دنیای واقعی اگر جسمی از ما فاصله بیشتری داشته باشد کوچکتر دیده می‌شود، این تعریف بسیار ساده و مختصه‌ی از perspective view است، نور بازتاب شده از اجسام به مرکز چشم ما برمی‌گردد و اجسامی که نزدیک‌تر هستند بزرگ‌تر دیده می‌شوند، در مقابل این مدل orthographic views قرار دارند، در این مدل که در طراحی‌های صنعتی بیشتر کاربر دارد و در دنیای واقعی وجود ندارد، هر کدام از pixel‌های صفحه پرتویی به صورت جداگانه از خود ساطع می‌کنند، همه این پرتوها موازی هستند و در نتیجه فاصله جسم از بیننده در نحوه دیدن شکل تاثیری ایجاد نمی‌کند، ما برای داشتن یک فضای سه بعدی واقع گرایانه از perspective projection استفاده می‌کنیم، برای ساختن این ماتریس از GLM به شکل زیر استفاده می‌کیم:

برنامه ۲۱.۱ fragment shader to use texture :

```
1 glm::mat4 proj = glm::perspective(  
2     glm::radians(45.0f),  
3     (float)width/(float)height,  
4     0.1f, 100.0f);
```

آرگومان اول مقدار `fov` (field of view) است، این مقدار به نحوی مقدار بزرگنمایی تصویر ما را مشخص می‌کند، آرگومان دوم aspect ratio دوربین و مقادیر سوم و چهارم به ترتیب فاصله صفحه نزدیک و دور را مشخص می‌کنند، این مشخصات تشکیل یه فضای سه بعدی می‌دهند که هر چه خارج از آن باشد قابل مشاهده نیست، همچنین دید perspective را به ما ارائه می‌کنند. شکل زیر بیانگر این موضوع است:

حالا که سه ماتریس `model`, `view` و `projection` را در اختیار داریم می‌توانیم با ضرب کردن مقدار `x`, `y`, `z`, `w` برای هر نقطه آن نقطه را به clip space ببریم، به ترتیب با ضرب کردن هر یک از ماتریس‌هایی که پیش تر گفت شد به `clip space` سپس `view space` و در آخر به `world space` رویم، همچنین `projection matrix` در



شکل ۷.۱ perspective frustum

نهایت تمامی مقادیر را به NDC نیز منتقل می کند.

مقدار تمامی ماتریس های بالا را با استفاده از متغیر های uniform vertex shader به vertex shader انتقال می دهیم، مقادیری از این سه ماتریس را که نیاز باشد در هر اجرا از render loop دوباره بارگزاری می کنیم vertex shader به شکل زیر در می آید:

برنامه ۲۲.۱ going 3D

```
1 #version 330 core
2
3 layout (location = 0) in vec3 aPos;
4 ...
5 uniform mat4 model;
6 uniform mat4 view;
7 uniform mat4 projection;
8
9 void main()
10 {
11     // note that we read the multiplication from right to left
12     gl_Position = projection * view * model * vec4(aPos, 1.0);
13 }
```

به صورت پیش فرض OpenGL نمی تواند مشخص کند که کدام قسمت از مدل هایی که رندر کرده در تصویر نهایی جلوتر یا عقب تر باید باشند، یعنی امکان دارد قسمت پشتی مکعب به جای قسمت جلوی آن رندر شود، در واقع هر کدام از fragment‌ها که دیر تر رندر شوند، بر دیگر fragment‌هایی که در آن پیکسل از قبل رندر شده اند پیروز می شوند و نمایش داده می شوند، برای حل این مشکل از z-buffer استفاده می کنیم، GLFW به صورت پیش فرض این buffer را برای ما ساخته، برای فعال سازی آن دستور زیر را پیش از render loop قرار می دهیم:

برنامه ۲۳.۱ enabling depth buffer

```
1 glEnable(GL_DEPTH_TEST);
```

و در انتهای render loop نیز دستور زیر را قرار می دهیم:

برنامه ۲۴.۱ \square
clearing depth buffer for next frame:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

خروجی به شکل زیر خواهد بود:



شکل ۸.۱ **cubes in 3D**

۴.۱ نورپردازی و سایه ها

به صورت کلی در این پروژه سه مدل light caster, directional و point پیاده سازی شده است که نور های spot می باشد.

۱.۴.۱ Directional Lights

به صورت ساده این گونه از نور را مشابه به نوری در نظر می گیریم که منع آن در بی نهایت قرار دارد، این فرض باعث می شود که تمامی پرتو های آن تقریبا با هم موازی باشند، خورشید را به همین صورت یک directional light در نظر می گیریم. برای بیان کردن این مدل از نور تنها نیاز به سه مولفه (x, y, z) داریم تا بتوانیم یک جهت را در فضای سه بعدی مشخص کنیم که همان جهت پرتو های نور می باشد.

انواع دیگر نور به این شکل ساده نیستند و از ویژگی های دیگری نیز برخوردارند، برای مثال point light دارای یک مولفه سه بعدی position است، همچنین برای این که attenuation یا میرایی نور را شبیه سازی کنیم از سه مقدار دیگر به عنوان quadratic, linear, constant نیز استفاده می کنیم تا بتوانیم تقریبا میرایی نور به نسبت فاصله از منبع نور را شبیه سازی کنیم. به همین صورت مقادیر دیگری از جمله direction را نیز داراست تا به کمک cut off بتواند یک نور که میرایی دارد و در جهت و زاویه خاصی در حال تاییدن است شبیه سازی کند.

۲.۴.۱ Lighting models

پیاده سازی و شبیه سازی رفتار واقعی نور کاری بسیار پیچیده است که قدرت سخت افزاری و هزینه زیادی را در بردارد، برای همین ما از روش هایی استفاده می کنیم که به صورت تقریبی بتوانند رفتار نور را به صورتی که ما از فیزیک می دانیم شبیه سازی کنند، برای شبیه سازی رفتار نور و برخورد آن با محیط از روش هایی مانند blinn-phong و lambertian استفاده می شود.

Lambertian

این روش بر اساس زاویه تابش نور با سطح جسم عمل می کند، بدینگونه که اگر سطح عمود به جهت نور باشد تقریبا تمام انرژی نور را دریافت می کند، اگر سطح به صورت موازی با جهت نور قرار داشته باشد مقدار انرژی دریافتی از منبع نور برابر صفر خواهد بود، بقیه حالات قرار گیری این دو نسبت به بسته به $\cos \theta$ مقدار دهی می شود. این روش مستقل از مکان تماشاگر است، در دنیای واقعی معمولاً بسته به material استفاده شده در جسم امکان بازتاب و بازتاب بیشتر نور از سطح در زاویه های به خصوصی وجود دارد، روش phong و بعد تر blinn این ویژگی را در خود جای دادند.

Blinn-Phong

این مدل متشكل از سه بخش اصلی است:

ambient: همانطور که در مدل lambertian دیدیم، اگر بردار نرمال سطحی عمود بر جهت نور باشد، آن سطح

هیچ مقداری از نور را دریافت نمی کند و کاملاً تاریک خواهد ماند، این حالت در جهان واقعی نیز در مکان های کاملاً بسته اتفاق می افتد و در دیگر حالات مقدار کمی نور از منابع مختلف باعث دیده شدن جسم می شوند، ما در این مدل مقداری از رنگ هر جسم را بدون توجه به زاویه تابش نور به آن جسم می دهیم، این باعث می شود که اجسام کاملاً تاریک به وجود نیایند، به این مقدار ambient color می گوییم.

diffuse: این مقدار برابر با همان مقداری است که در مدل lambertian نیز محاسبه کرده بودیم، بسته به جهت

بردار نرمال سطح و جهت نور مقداری انرژی توسط هر سطح جذب می شود و باعث روشن شدن سطح می گردد.

specular: این مقدار باعث به وجود آمدن درخشندگی و بازتاب بیشتر نور نسبت به مکان ناظر می شود، این مقدار

مستقل از مکان ناظر نیست، اگر بردار view direction در جهت بردار بازتاب نور باشد از حداقل درخشندگی و
highlight برخوردار می شود، برای محاسبه کردن جهت بازتاب از تابع reflect در GLSL استفاده می کیم، سپس برای محاسبه مقدار بازتاب از ضرب داخلی بین بردار reflectDir و view direction استفاده می کنیم، همچنین برای اینکه از منفی شدن مقادیر جلوگیری کنیم از تابع max نیز استفاده می کنیم:

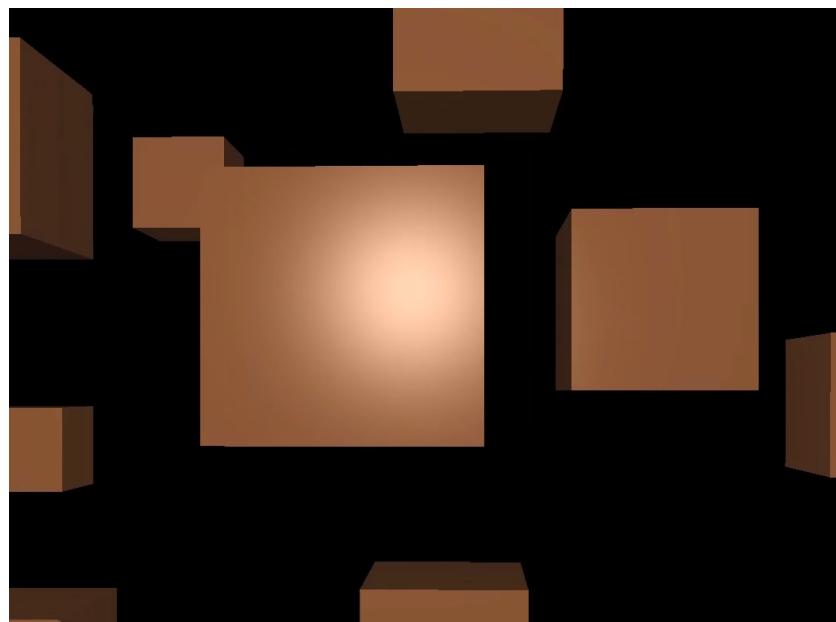
برنامه ۲۵.۱ calculating specular component :

```
1 vec3 viewDir = normalize(viewPos - FragPos);  
2 vec3 reflectDir = reflect(-lightDir, norm);  
3 float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
```

مقدار ۳۲ در کد بالا را مقدار shininess می گویند، هر چقدر این مقدار بیشتر باشد درخشش بازتاب در آن نقطه بیشتر می شود.

در نهایت با جمع بستن این سه مقدار می توانیم تقریبی از رفتار نور در دنیای واقعی داشته باشیم، در این پروژه از روش blinn-phong استفاده شده. نتیجه استفاده از این روش به شکل زیر است:

در حال حاضر تمام سطوح مقداری که بازتاب می کنند تنها به میزان زاویه با منبع نور بستگی دارند، برای این که بتوانیم این بازتاب را کنترل کنیم و سطوحی با میزان بازتاب متفاوت را تشکیل دهیم، در اینجا نیاز به جزئیات بیشتر نسبت به یک سطح داریم، همانگونه که قبل تراز texture ها برای ایجاد جزئیات روی سطوح استفاده کردیم این بار نیز از نوع دیگری از تکسچر ها برای اینکار استفاده می کنیم، به این تکسچر ها specular map می گوییم، برای گرفتن مقادیر برای هر پیکسل دقیقاً مانند texture ها کار می کنیم، یک نمونه از specular map به شکل زیر است:



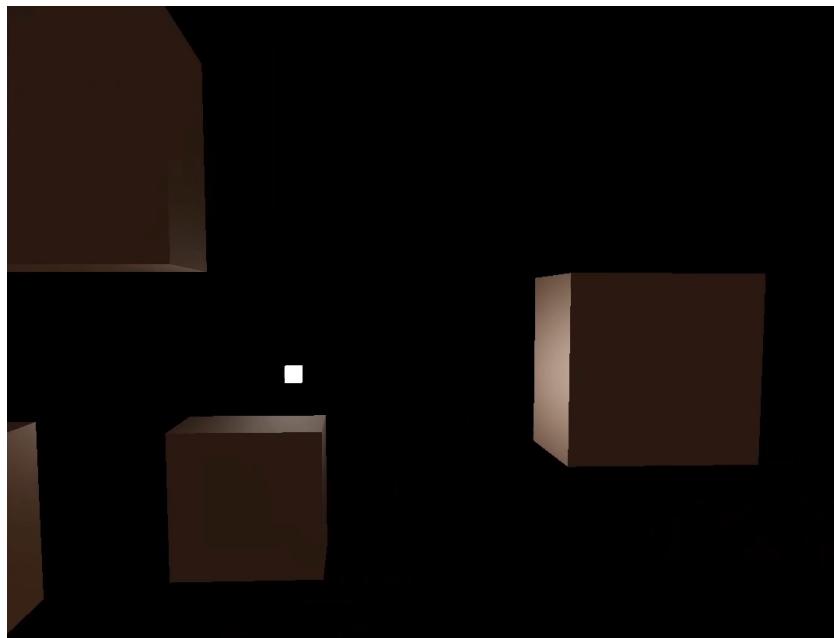
شکل ۹.۱ blinn-phong shading

بعد از استفاده از این specular map برای مدل اسلحه نتیجه به شکل زیر در می آید، می بینید که مقدار یازتاب

در همه نقاط یکسان نیست:



شکل ۱۲.۱ a gun specular map

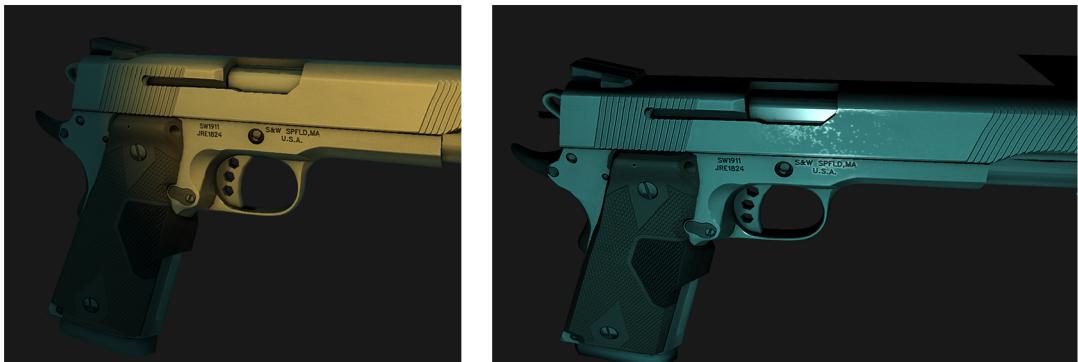


شکل ۱۰.۱ blinn-phong shading

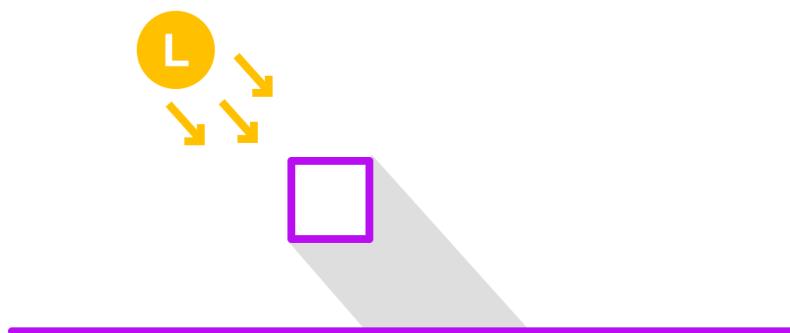
Shadows ۳.۴.۱

الگوریتم های زیادی برای تولید سایه وجود دارند، برخی از این الگوریتم ها مناسب real-time نیستند و به همین دلیل مورد بحث ما نیستند، این الگوریتم ها غالباً از روش های ray-tracing سرچشمه می‌گیرند، در الگوریتم های ray-tracing به صورت خود کار سایه ها تولید می‌شوند، دلیل آن نیز نرسیدن پرتو های نور به نقاط است.

روش هایی که برای تولید سایه در real-time استفاده می‌کنیم شباهت زیادی از لحاظ تعریف به روش بالا دارند، باید سطوحی را پیدا کنیم که نور به آن ها نمی‌رسد، یکی از این روش ها shadow mapping نام دارد، این روش و مشتقات آن در بازی کامپیوتری مورد استفاده قرار می‌گیرند. shadow mapping روشهای ساده است که در مراحلی که در زیر شرح می‌دهم قابل تولید است: اساس کار ایجاد نقشه‌ای است که با استفاده از آن می‌توانیم مشخص کنیم که نور به چه جسمی زودتر برخورد کرده و چه جسمی به نور نزدیک‌تر است



شکل ۱۱.۱: phong shaded gun



شکل ۱۳.۱: absence of light - shadow

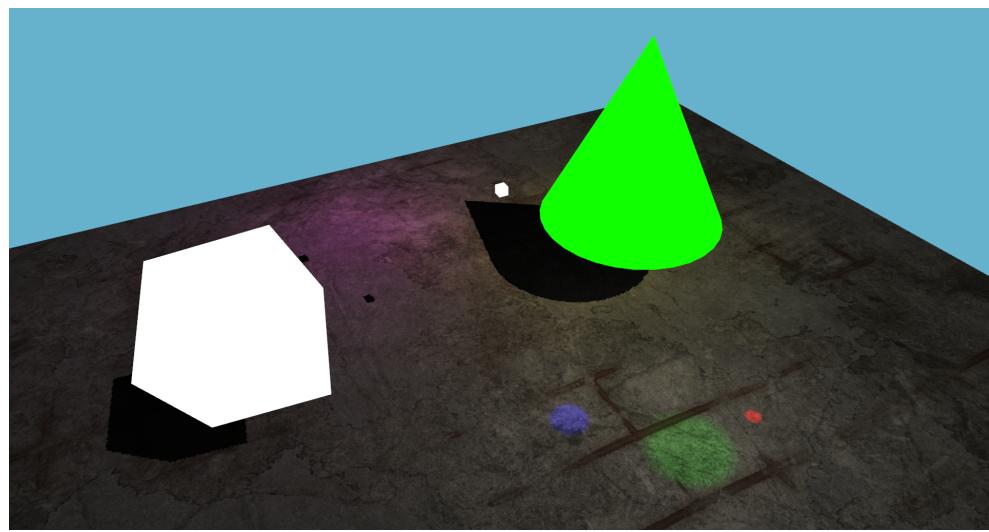
همانگونه که در شکل بالا نیز پیداست، سایه به دلیل نرسیدن نور و غیاب نور در نقطه پدید می‌آید، در روش shadow mapping ابتدا صحنه را از دید منبع نور به صورت ساده رندر می‌کنیم، در این عملیات ما depth buffer را خالی می‌گذاریم، زیرا نیازی به خروجی آن نداریم، تنها کاری که باید انجام شود پر شدن fragment shader است، این کار به صورت خودکار انجام می‌شود پس ما نیازی به نوشتمن کد در fragment shader نداریم، depth buffer تشکیل شده را به یک تکسچر متصل می‌کنیم، به این صورت shadow map ما آماده است.

بعد از اینکه shadow map آماده شد، حالا صحنه را از اول و با تمام جزئیات موردنیاز رندر می‌کنیم، و در تصمیم می‌گیریم که آیا این نقطه نزدیک ترین نقطه در برخورد با پرتو نور است یا خیر، برای اینکار نیاز داریم که نقطه‌ای که در حال پردازش هست را به light space ببریم، این کار در مرحله vertex shader انجام می‌شود، مقدار این ضرب به مرحله fragment shader فرستاده می‌شود و توسط light Space Matrix آن جا به شکل زیر برای محاسبه نور استفاده می‌شود:

برنامه ۲۶.۱ calculate shadow

```
1 float ShadowCalculation(vec4 fragPosLightSpace)
2 {
3     // perform perspective divide
4     vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
5     // transform to [0,1] range
6     projCoords = projCoords * 0.5 + 0.5;
7     // get closest depth value from light's perspective (using [0,1]
8         range fragPosLight as coords)
9     float closestDepth = texture(shadowMap, projCoords.xy).r;
10    // get depth of current fragment from light's perspective
11    float currentDepth = projCoords.z;
12    // check whether current frag pos is in shadow
13    float shadow = currentDepth > closestDepth ? 1.0 : 0.0;
14
15 }
```

در تابع بالا ابتدا مقادیر $[x, y, z]$ تقسیم بر مقدار w می شوند زیرا light space matrix به صورت orthogonal محسوب شده بود، زیرا همانطور که پیش تر گفته شد نور با یکدیگر موازی هستند و اگر بخواهیم از دید منبع نور جهان را رندر کنیم باید از view matrix استفاده کنیم که orthogonal باشد، حالا اما برای استفاده در رندر نهایی نیاز به perspective view داریم، برای همین این تقسیم را انجام می دهیم. در مرحله بعد مقادیر را به بازه $[0 - 1]$ می آوریم، از پیش میدانیم که مقادیر بین $[1 - 1]$ هستند پس با تقسیم کردن به دو و جمع با 0.5 به بازه دلخواه ما منتقل می شوند. در مرحله بعد مقدار closest depth را از تکسچری که در مرحله قبل رندر کرده بودیم دریافت می کیم، این مقدار، نزدیک ترین جسم به نور است، در خط بعد این مقدار را با عمق نقطه فعلی در نظر مقایسه می کنیم، اگر عمق فعلی از نزدیک ترین عمقی که داریم بیشتر باشد پس این پیکسل در سایه است پس مقدار 1.0 به معنی True برمی گردانیم. حاصل کارهای بالا به ساختن سایه منجر می شود:



شكل ١٤.١ scene with shadow

۵.۱ کاربردها

Use Cases ۱.۵.۱

به پروسه تولید یک تصویر از داده ای سه بعدی گفته می شود، rendering real time rendering اشاره به تولید متنابض تصاویر بر اساس حرکت دوربین یا داده های سه بعدی دارد. این گونه از Renderer ها در تمامی صنایعی که نیاز به تولید تصاویر سه بعدی و هر گونه شبیه سازی داشته باشند به دلیل هزینه پایین تر برای ساختن هر frame مورد استفاده قرار می گیرند.

برای مثال گوشه ای از کاربردهای این نرم افزارها را در موتور Eevee می بینیم. Eevee موتور رندر سه بعدی realtime استفاده شده در نرم افزار blender است، Eevee به منظور افزایش سرعت در زمان توسعه و گاهای در محصولات تجاری عرضه شده است. در زمان کار با blender و در صحنه های بزرگ و شلوغ با استفاده از Eevee می توانیم با سرعت بالا دسترسی به اکثر جزئیات و ویژگی های صحنه خود داشته باشیم، اینکار بدون استفاده از یک Eevee real time renderer پروسه ای بسیار زمان برو شامل آزمون و خطاهای زیادی می باشد که هزینه هایی زیادی از نظر مالی و نیروی انسانی به جای می گذارد، Eevee به صورت مأذولات و منطبق با قراردادهای Blener ساخته شده. همچنین Eevee با استفاده از OpenGL و تکنیک های render استفاده شده در صنعت بازی سازی موفقیت های زیادی کسب کرد.

متوتری که من در این پژوهه توسعه دادم از لحاظ نرم افزاری در دسته بندی نرم افزارهایی مانند Eevee قرار می گیرد، Shm مانند Eevee تمرکز بر تولید سریع و شبیه سازی مطابق با واقعیت تصاویر دارد، اجزا زیادی مانند:

۱. ارائه دو مدل دوربین perspective و ortho

۲. ارائه سه نوع کلی از light effect مانند point lightcaster، directional or sun و spot

۳. تولید shadow و highlight

۴. component هایی برای کنترل تصاویر، textures و gpu buffer

۵. کنترل خودکار و دستی uniform buffer objects

۶. ارائه کلاس Model برای load کردن مدل های سه بعدی و مدیریت کردن material ambient and texture maps و ویژگی های آن ها

۷. ارائه کلاس ها برای `load` و `compile` کردن `shader` ها

۸. و کلاس های عمومی تر برای بیان کردن رنگ ها و ساختار های ریاضی مانند `vector` های سه و چهار بعدی،
Quaternion های سه در سه و چهار در چهار، Matrix

... ۹

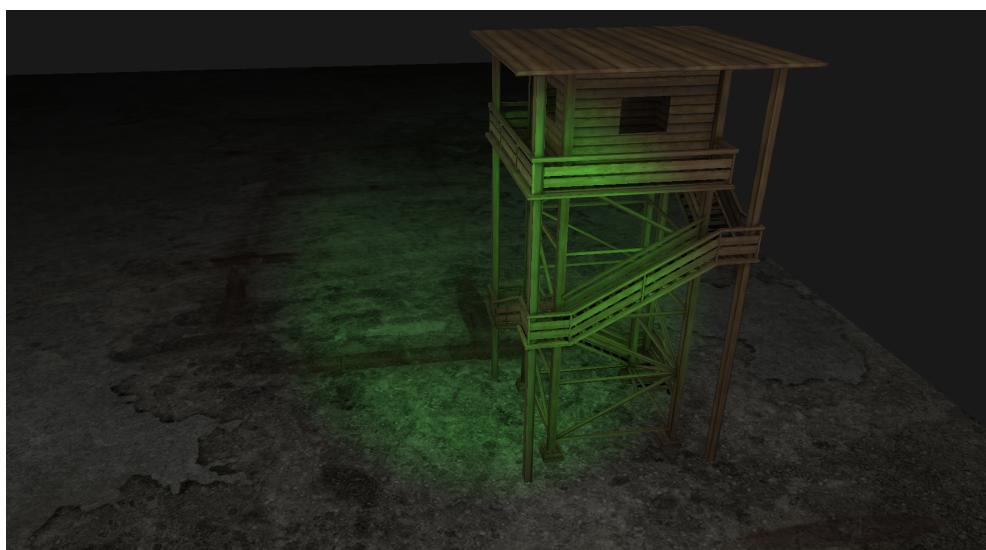
توسط `shm` ارائه می شود.

3D viewer ۲.۵.۱

این پروژه قابل استفاده برای Microsoft 3D viewer 3D scene viewer ها مانند است:



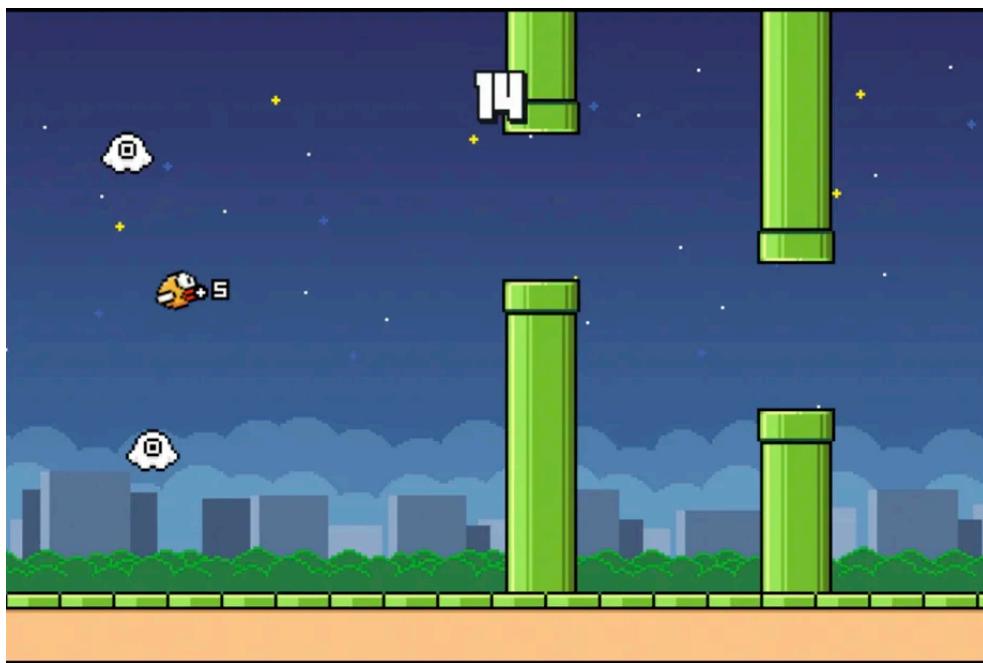
شکل ۱۵.۱ model in microsoft 3d viewer



شکل ۱۶.۱ model in my project wit a green point light and ground

Games ۳.۵.۱

یکی از کاربردهای real-time render engine که گفته شد، در صنعت بازی سازی است، در ادامه بازی معروف Flappy bird که یک بازی دو بعدی و Arcade است را در با اشیاء سه بعدی بازسازی می کنیم، در این بازی باید پرنده را با فشار دادن درست و بجای space از بین pipe ها عبور دهیم:



شكل ۱۷.۱ original flappy bird

برای شروع ابتدا مدل های pipe و bird را در blender می سازیم، سپس یک فایل جدید ایجاد می کنیم و را به وسیله کد زیر به آن اضافه می کنیم:

برنامه ۲۷.۱: adding Engine to project

```
1 #include "Engine.hpp" // include needed component and classes for rendering  
2 #include "Light.hpp" // if you want light caster in scene, include them
```

دو shader متفاوت برای پرنده و لوله ها می سازیم:

برنامه ۲۸.۱: creating shaders

```

1 std::shared_ptr<shader> pipeshader = Engine::CreateShader("./assets/
    shaders/model_loading.vs", "./assets/shaders/model_loading.fs");
2 std::shared_ptr<shader> birdshader = Engine::CreateShader("./assets/
    shaders/model_loading.vs", "./assets/shaders/model_loading.fs");

```

همچنین قابلیت استفاده از یک shader برای هر دو شی در بازی وجود دارد.

دو تابع ویژه حتما باید در این فایل تعریف شوند، تابع های `inLoop` و `outLoop` را در همین فایل تعریف می کنیم، وظیفه این تابع ها از نام آن ها پیداست، تابع `main render loop` در `inLoop` بازی صدا زده می شود، پس ویژگی های متغیر و مورد نیاز که در هر frame باید اجرا شوند را در این تابع می نویسیم، کار تابع `outLoop` این است که تنظیمات و اعمالی که قبل از شروع `render loop` انجام می شود را انجام دهد، برای مثال ما فقط یکبار نیاز به `load` کردن مدل ها در بازی داریم و آن هم قبل از اجرای `render loop` است، پس به شکل زیر مدل هارا در بازی `load` می کنیم.

برنامه ۲۹.۱

```

1 Engine::getRenderer()
2     ->LoadModel(
3         "./assets/models/texturedbird.obj", // path to 3D
4             model
5             birdshader
6         );

```

با استفاده از توابع موجود در `Model` می توانیم `scale`, `position`, `rotation` مدل را تغییر دهیم، اینکار را برای `bird` به شکل زیر انجام می دهیم تا در موقعیت شروع بازی قرار بگیرد:

برنامه ۳۰.۱

```

1 GET_MODEL(bird)->setPosition({-3.0f, 5.7f, 0.0f});
2 GET_MODEL(bird)->setScale({0.5f, 0.5f, 0.5f});
3 GET_MODEL(bird)->setRotation({0.0f, 1.0f, 0.0f}, 45.0f);

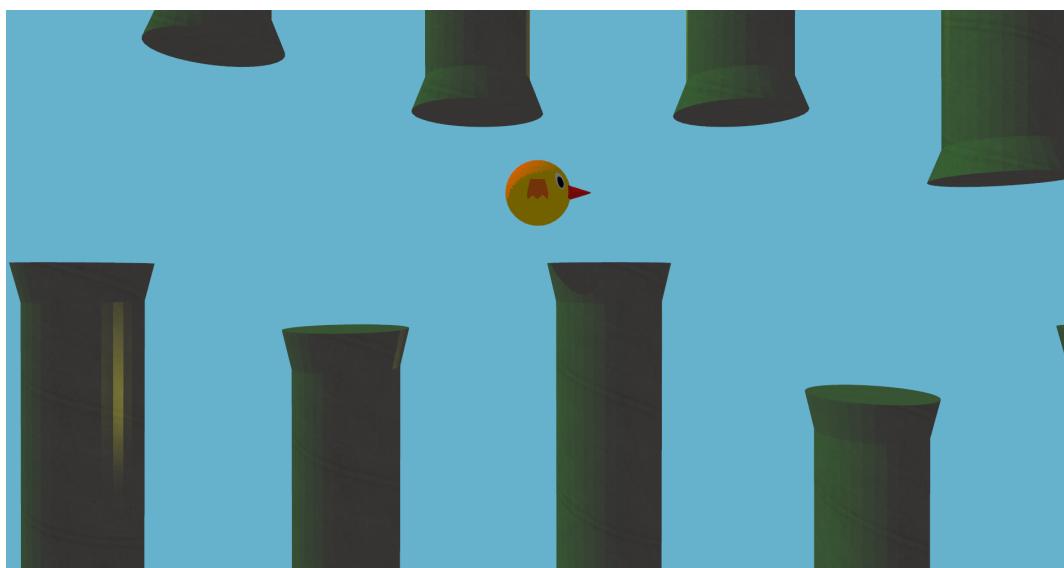
```

مدل مربوط به `pipe` را به شکل بالا `load` می کنیم، چون `pipe` به تعداد زیادی در محیط بازی قرار دارد، آن را با استفاده از دستور `DrawInstances` در تابع `inLoop` در محیط بازی رسم می کنیم:

برنامه ۳۱.۱ \square draw instances of a model

```
1 GET_MODEL(0)->DrawInstances(pipes_pos, nullptr);
```

در تابع بالا، آرگومان اول آرایه ای از `glm::vec3` است که بیانگر مشخصات قرار گیری هر کدام از `pipe` ها است، این مقادیر در تابع `outLoop` به وسیله یک حلقه تولید شده اند. برای بررسی برخورد `bird` و `pipe` ها در بازی از تابع زیر استفاده می کنیم، اگر پرنده با یکی از لوله ها برخورد کند به حالت `sleep` تغییر وضعیت می دهد و فیزیک بازی تاثیری بر آن ندارد، از این حالت برای `reset` کردن بازی استفاده می کنیم. تصویری از محیط در حال اجرای بازی به شکل زیر است:



شکل ۱۸.۱ my project flappy bird

برای دانلود بازی می توانید از این لینک استفاده کنید، همچنین برای دیدن فیلم اجرای بازی می توانید از این لینک استفاده کنید.