



دانشکده فنی و مهندسی

پروژه کارشناسی رشته مهندسی کامپیوتر

عنوان: پیاده سازی real-time 3D engine

استاد راهنما: سرکار خانم دکتر قاسمیان

دانشجو: احمد منصوری بیدخوانی

تابستان ۱۴۰۰

چکیده

همزمان با پیدایش کامپیوتر ها، تلاش ها برای بهره بردن از توان آنها برای ابزار های Visualize و دیگر ابزار ها برای استفاده از این قابلیت در زمینه های نظامی، فیلم و انیمیشن، شبیه سازی و بازی سازی شروع شد، این ابزار ها یا به صورت اختصاصی برای استفاده در صنایع خاص طراحی می شوند یا به صورت عمومی تر برای کاربرد های وسیع تری طراحی و توسعه می یابند. طراحی و پیاده سازی موتور های گرافیکی به صورت کلی دارای پایه ها و دانشی یکسان از نحوه کار پردازنده ها و ریاضیات است.

این گزارش حول محور ساخت یک real-time render engine نوشته شده که سعی در ارائه خلاصه ای از نحوه ساختن این نرم افزار می باشد. render engine ها پایه های اصلی تمامی شبیه ساز ها، ابزار های صنعتی، بازی های کامپیوتری و ابزار آلات 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.

فهرست مطالب

۴

اول گرافیک

۶

Renderer ۱

۷

پنجره ها و کانتکست ها

۹

تصویر کردن داده ها

۲۸

دوربین و حرکت در جهان

۳۴

نورپردازی و سایه ها

۴۲

کاربردها

بخش اول

گرافیک

مقدمه

در بخش اول به معرفی و توضیح قسمت های مربوط به تصویر در پروژه می پردازم، این قسمت از موتور مسئولیت دریافت مدل های سه بعدی و اطلاعات مربوطه (**Textures, Normals**، ...) نمایش آن ها در صفحه را بر عهده دارد، در این قسمت ما با استفاده از ریاضیات مدل ها در فضای سه بعدی شبیه سازی می کنیم.

تمامی ابزار های استفاده شده در این قسمت، در طول فصول و متناسب با بخشی که از آن ها استفاده شده معرفی می شوند.

هر فصل در این بخش مستقیماً مربوط به یکی از قسمت های موتور در بخش گرافیک است، ابتدای هر فصل فایل های مربوطه به آن فصل ذکر خواهند شد.

همچنین از طریق این لینک به مخزن پروژه دسترسی خواهید داشت.

فصل ا

Renderer

پنجره ها و کانتکست ها

openGL

یک *api* چند زبانه و کراس پلتفرم است که برای به تصویر کشیدن تصاویر دو بعدی و سه بعدی با استفاده از بردارها اسفاده می شود، معمولا 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(*library*) اشتباہ گرفت، *OpenGL* به صورت *interface* و یک قراداد انتزاعی در ورژن های مختلف ارائه می شود که فروشندهان و سازندگان(*vendor*) مختلف پیاده سازی ای منطبق با این قرارداد 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.

GLFW

یک کتابخانه برای ساختن *window* و *context* ها برای *openGL*, *openGL ES*, *Vulkan* 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* نیاز به این کتابخانه یا مشابه آن داریم زیرا *openGL* هیچگونه قابلیت پیشفرضی برای مدیریت *window* یا *context* ها یا مدیریت *input* ندارد. همچنین GLFW یک کتابخانه چندسکویی است و می توانیم آن را در سیستم عامل های مختلف استفاده کنیم، پروژه من نیز چندسکویی است، پس می توانیم از این کتابخانه سبک و چندسکویی استفاده کنیم.

GLFW از window، پنجره ای است که به وسیله امکانات فراهم شده در سطح سیستم عامل برای ما فراهم می کند، همچنین یک *context* را می توانیم به عنوان یک شئ در نظر بگیریم که تمامی اطلاعات *openGL* را به همراه دارد، اطلاعاتی مانند *framebuffers* و *state* ها. برای کنترل کردن ورودی ها، *glfw* از دو روش استفاده می کند، برای ورودی *mouse* و *keyboards* از *callback function* ها استفاده می کند، اما برای ورودی *keyboards* می توانیم از تابع های کتابخانه 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.



شكل ۱.۱ : *glfw logo*

GLAD

کتابخانه ای برای *load* کردن *pointer* ها به توابع *opengl* در هنگام *runtime*. این کتابخانه یکی از کتابخانه های OpenGL Loading Library است، برای کار با *opengl* ما حتما باید یکی از این کتابخانه هارا مورداستفاده قرار دهیم تا بتوانیم به توابع *opengl* مشخص دسترسی داشته باشیم، این کتابخانه ها هم ویژگی های Core که توسط *opengl* مشخص شده را *load* می کنند و هم ویژگی های extension 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 پaramتر هایی که کاربر انتخاب می کند یک فایل حاوی تمامی تعریف های مربوط به *constant* و *تابع* 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 *opengl function poiter* هارا در *runtime* بارگزاری کنیم.

برنامه ۱.۱ \square
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.

تصویر کردن داده ها

Vertex Data

برای *render* کردن تصاویر نیاز به اطلاعاتی داریم، با مثلث شروع می کنیم، مثلث در گرافیک کامپیوترا 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 شرابط به همین شکل است، ما نیاز به سه نقطه داریم، تفاوت این نقاط با نقطه های روی صفحه در ابعاد آن است، نقاط روی صفحه دو بعدی بودند، `OpenGL` نقاط را به صورت سه بعدی دریافت می کند، هر کدام از این نقاط متشکل از سه مقدار برای x, y, z هستند، این نقاط را می توان به صورت بردار هایی در *Normalized Device Coordinates* نمایش داد، یک بردار در C را به شکل رو به رو نمایش می دهیم:

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

مقادیر z 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 };

```

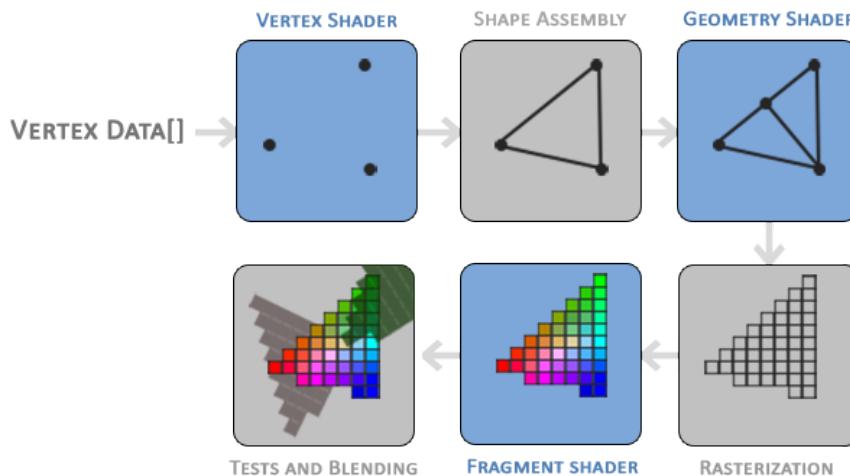
در قطعه کد بالا از ۹ عدد که همگی در بازه مشخص شده برای NDC هستند استفاده کردیم، توجه کنید که اعداد در یک آرایه و یه صورت پشت سر هم به برنامه داده شده اند، هیچگونه جداسازی یا طبقه بندی بر اساس نقاط مختلف صورت نگرفته و همچنین این مقادیر هنوز بر روی GPU آپلود نشده اند، دسته بندی کردن این اطلاعات خام و آپلود 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, ...

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

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

Shader 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 دهدند، بر روی کارت گرافیک های امروزی تمامی shader ها به جز دو نوع از آن ها به صورت پیشفرض وجود دارد، این دو هستند که حتما باید توسط برنامه نویس به کارت Fragment shader و Vertex shader گرفته شوند.



Graphic pipeline: from learnopengl.com : 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 فضای دیگر از ماتریس ها استفاده می شود، به طوری که هر فضا دارای یک Transformation Matrix است، برای رسم مثلث مانیازی به تغییر فضانداریم و در NDC ادامه می دهیم. یک Vertex shader ساده به صورت زیر است:

برنامه ۱: basic vertex shader

```

1 #version 330 core
2 layout (location = 0) in vec3 aPos;
3
4 void main()
5 {
6     gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
7 }
```

کد بالا ساده‌ترین مدل برای استفاده از vertex shader است، این کد به زبان GLSL(OpenGL Shading Language) نوشته شده است، در خط اول ما نوع ویژگی‌ها را که پیش‌تر توضیح داده بودیم را مشخص کرده‌ایم، خط ۲ یک متغیر از نوع vec3 به اسم aPos تعریف کرده‌ایم، با استفاده از *layout (location = 0)* در این خط مشخص کرده‌ایم 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 نشان‌دهنده مکان این نقطه که در حال پردازش است می‌باشد را به وسیله مقادیری که از کد C خواندیم و مقدار چهارم که ۱ مقداردهی کردیم، مقدار چهارم در این فضای کاربردی ندارد اما در فضای سه بعدی و در perspective view به کار می‌آید.

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

RENDERER . فصل ۱

برنامه ۱.۱ 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 vec4 از نوع FragColor تعريف شده و مقادیری که دریافت کرده به ترتیب معنی red, green, blue, alpha را می دهند، مقادیر باید 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 shader program link و compile کردن را نشان می دهد.

برنامه ۱.۲ 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
```

۱۶

```
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
```

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

Upload Data to GPU

برای استفاده از مشخصات نقاطی که تعریف کردیم باید آن هارا روی کارت گرافیک آپلود کنیم، انتقال اطلاعات بین gpu و cpu 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:

برنامه ۶.۱: creating a buffer object

```

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

```

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

برنامه ۷.۱: binding to target gl_array_buffer

```
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 متنقل کنیم:

برنامه ۸.۱: uploading data for static draw

```

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 ها در vertex shader باید نوع تفسیر داده هارا نیز به صورت دستی مشخص کنیم، به اینکار Linking Vertex Attribute می گویند.

```
float vertices = [
```

<i>a vertex</i>	- 0.5 , - 0.5 , 0.0 ,
<i>a vertex</i>	0.5 , - 0.5 , 0.0 ,
<i>a vertex</i>	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

مشخص کردیم که در این مورد برابر با صفر است. وسیله (location = 0)

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` می گویند.

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.

شکل ۱. ۳. متناسب با توضیحات ساخته شده.

برنامه ۹.۱ link vertex attribute to vertex data :

```

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 VAO را bind 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,
               GL_STATIC_DRAW);
6 // 3. then set our vertex attributes pointers
7 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
                          (void*)0);
8 glEnableVertexAttribArray(0);

```

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

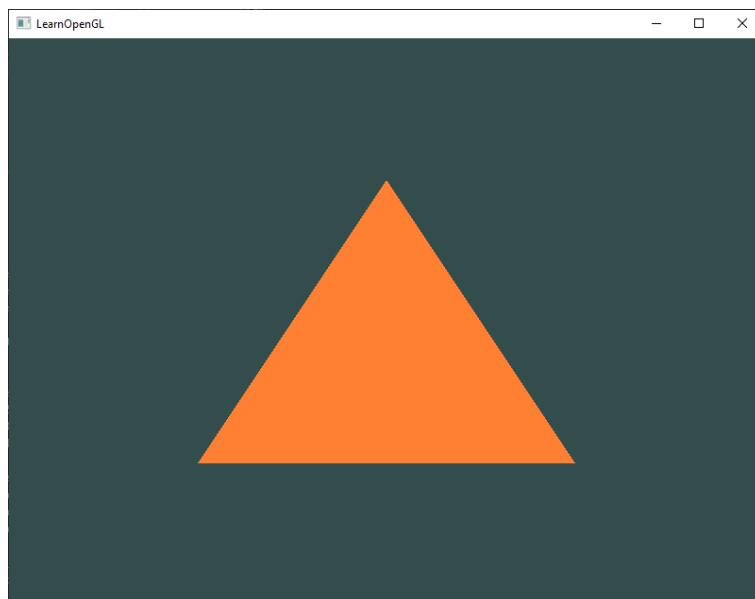
Render Loop

حالا برای رسم کردن مثلث نیاز به کدهای زیر داریم:

link vertex attribute to vertex data : ۱۱.۱ برنامه

```
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، حال تمامی داده ها و تفسیر ها آماده هستند، برای رسم ازتابع 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.



شكل ۱.۴: 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 افزودن رنگ به هر کدام از vertex های attribute 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);
```

RENDERER .۱ فصل

```
8 }
```

حال 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:

position and color data in one array : ۱۴.۱ برنامه

```
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 VAO را bind می کنیم، یک مرحله جدید در link vertex attribute برای رنگ ها اضافه شده است، این مرحله به شکل زیر است:

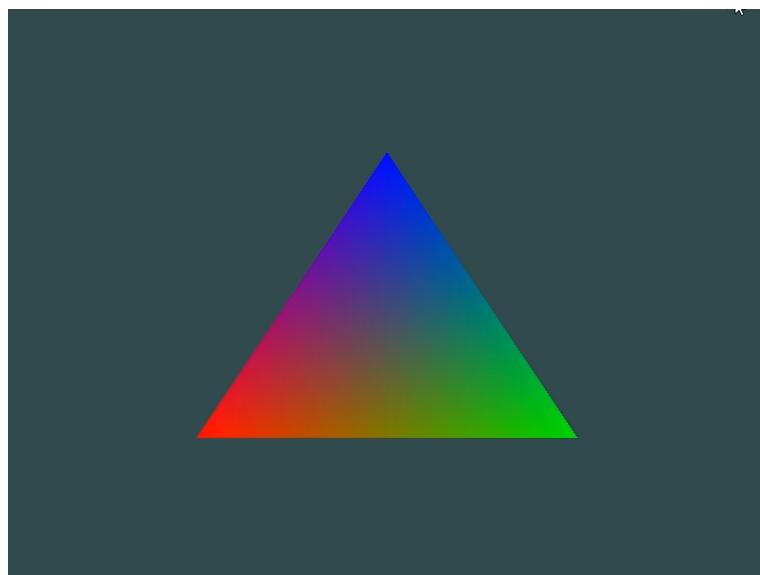
position and color data in one array : ۱۵.۱ برنامه

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

همانطور که می بینیم در این مرحله متغیر های مربوط offset و stride به شکل جدید

برای استفاده از و تغییر 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:



شکل ۱.۵: colorful triangle

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

Textures

تصور کنید می خواهیم یک دیوار 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 مربع یا مستطیل(دو مثلث که از وتر بر روی یکدیگر قرار گرفته اند) بسته به موقعیت نقاط می دهند. 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 و opengl linear interpolation که بین این رنگ texture ها انجام می دهد یک دیوار آجری رندر کنیم، راه حل این مشکل استفاده از یک opengl است، به صورت ساده می توانیم عکس یک دیوار آجری را به texture آپلود کرده به ازای هر fragment از یکی از pixel های عکسی که ما به عنوان texture آپلود کردیم 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 کردن آن بر روی Gpu.

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

۴. استفاده از توابع GLSL برای sample کردن texture

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

position and color data in one array : ۱۶.۱ برنامه

```

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.

حال 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 بازکردن 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:

load and upload data to gpu : ۱۷.۱ برنامه

```

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
6 // variable
6 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,

```

RENDERER . فصل ۱

```

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:

set minimum setting for a texture : ۱۸.۱

```

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 attribute به صورت یک vertex shader جدید تعریف می کنیم و به مرحله fragment shader می فرستیم، ارسال این داده ها بین مراحل مختلف pipeline را با استفاده از متغیر هایی که با *in*, *out* تعریف 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* تعریف می کنیم و در مرحله بعد آن را با *in* تعریف می کنیم، دقت کنید باید نام متغیر دقیقا برابر باشد، کد زیر به vertex shader اضافه می کنیم:

vertex shader to use texture : ۱۹.۱

```

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 می‌کند:

fragment shader to use texture : ۲۰.۱ برنامه

```

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 می‌آید:



شكل ٦.١ : textured triangle

دوربین و حرکت در جهان

Transformations

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 شبیه 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 translation 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 بردہ ایم.

برای انجام عملیات ریاضی مربوط به OpenGL بر روی CPU از کتابخانه GLM استفاده می کنیم، این کتابخانه دارای کلاس ها و توابعی است که نمایانگر ساختار های ریاضیاتی مانند vector ها و matrix ها است، همچنین تعداد دیگری از توابع را دارد است که برای ساده

سازی کار ما فراهم شده اند، یکی از این توابع `glm::LookAt` است، می توانیم از این کلاس برای تشکیل یک `view matrix` استفاده کنیم که نشانگر **Camera** است.

Camera

در دنیای واقعی اگر جسمی از ما فاصله بیشتری داشته باشد کوچکتر دیده می شود، این تعریف بسیار ساده و مختصری از `perspective view` است، نور بازتاب شده از اجسام به مرکز چشم ما برمیگردد و اجسامی که نزدیک تر هستند بزرگتر دیده می شوند، در مقابل این مدل `orthographic views` قرار دارند، در این مدل که در طراحی های صنعتی بیشتر کاربر دارد و در دنیای واقعی وجود ندارد، هر کدام از `pixel` های صفحه پرتویی به صورت جداگانه از خود ساطع می کند، همه این پرتوها موازی هستند و در نتیجه فاصله جسم از بیننده در نحوه دیدن شکل تاثیری ایجاد نمی کند، ما برای داشتن یک فضای سه بعدی واقع گرایانه از `perspective projection` استفاده می کنیم، برای ساختن این ماتریس از GLM به شکل زیر استفاده می کنیم:

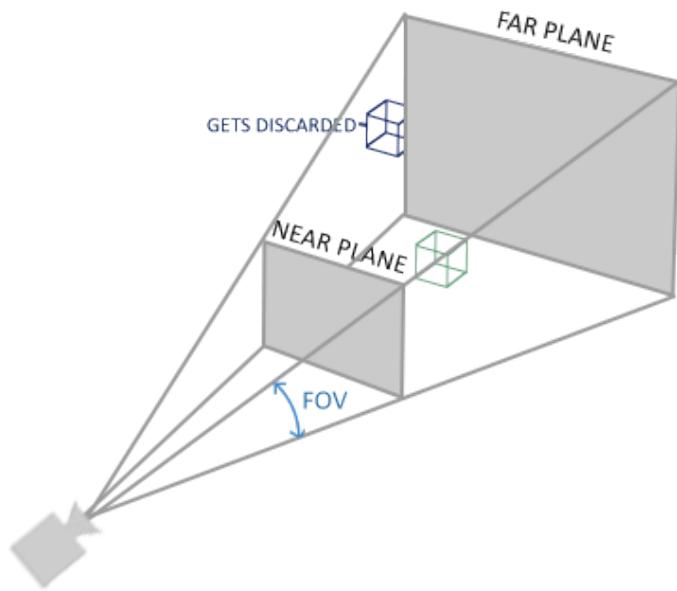
برنامه ۲۱.۱ \square
fragment shader to use texture

```

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 می کند، آرگومان دوم `aspect ratio` دوربین و مقادیر سوم و چهارم به ترتیب فاصله صفحه نزدیک و دور را مشخص می کنند، این مشخصات تشکیل یه فضای سه بعدی می دهند که هر چه خارج از آن باشد قابل مشاهده نیست، همچنین دید `perspective` را به ما ارائه می کند. شکل زیر بیانگر این موضوع است:



شکل ۱: perspective frustum :۷.۱

حالا که سه ماتریس projection و view را در اختیار داریم می توانیم با ضرب کردن مقدار ۱ برای هر نقطه آن نقطه را به clip space ببریم، به ترتیب با ضرب کردن هر یک از ماتریس هایی که پیش تر گفت شد به view space سپس view space و در آخر به clip space می رویم، همچنین projection matrix در نهایت تمامی مقادیر را به NDC نیز منتقل می کند.

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

RENDERER . فصل ۱

برنامه ۲۲.۱ : going 3D

```

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

```

به صورت پیشفرض OpenGL نمی‌تواند مشخص کند که کدام قسمت از مدل‌هایی 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 دیرer تر رندر شوند، بر دیگر fragment‌هایی که در آن پیکسل از قبل رندر شده اند پیروز می‌شوند و نمایش داده می‌شوند، برای حل این مشکل از z-buffer استفاده می‌کنیم، GLFW به صورت پیشفرض این buffer را برای ما ساخته، برای فعال سازی آن دستور زیر را پیش از render loop قرار می‌دهیم:

برنامه ۲۳.۱ : enabling depth buffer

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

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

clearing depth buffer for next frame : ۲۴.۱ برنامه

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

خروجی به شکل زیر خواهد بود:



شکل ۸.۱ cubes in 3D

نورپردازی و سایه ها

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

Directional Lights

به صورت ساده این گونه از نور را مشابه به نوری در نظر می گیریم که منبع 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 یا میرایی نور را شبیه سازی کنیم از سه مقدار دیگر به عنوان constant، attenuation و quadratic یا linear 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 تابیدن است شبیه سازی کند.

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 قرارگیری 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 جشم امکان بازتاب و بازتاب بیشتر نور از سطح در زاویه های به خصوصی وجود دارد، روش phong و blinn این ویژگی را در خود جای دادند.

Blinn-Phong

این مدل مت Shank 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 کنیم، سپس برای محاسبه

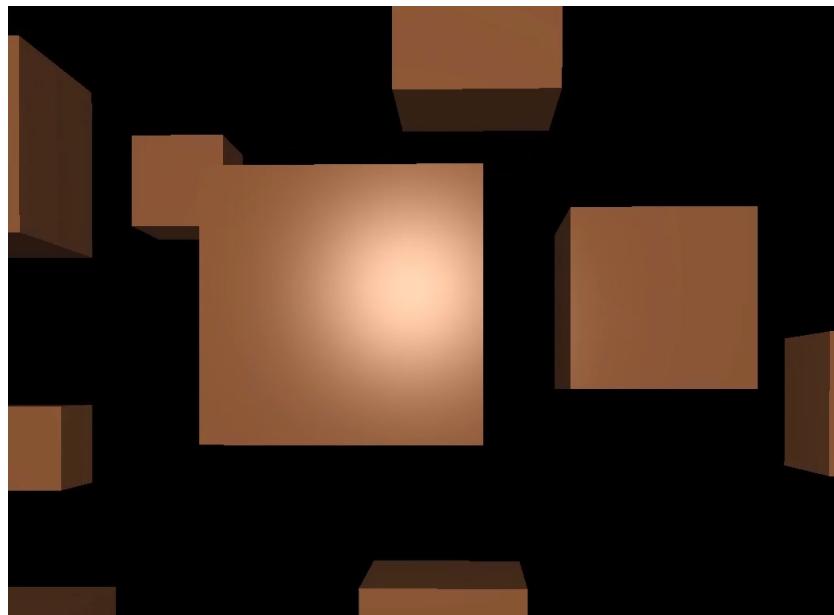
مقدار بازتاب از ضرب داخلی بین بردار view direction و reflectDir استفاده می‌کنیم، همچنین برای اینکه از منفی شدن مقادیر جلوگیری کنیم از تابع max نیز استفاده می‌کنیم:

برنامه ۲۵.۱ \square
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);
```

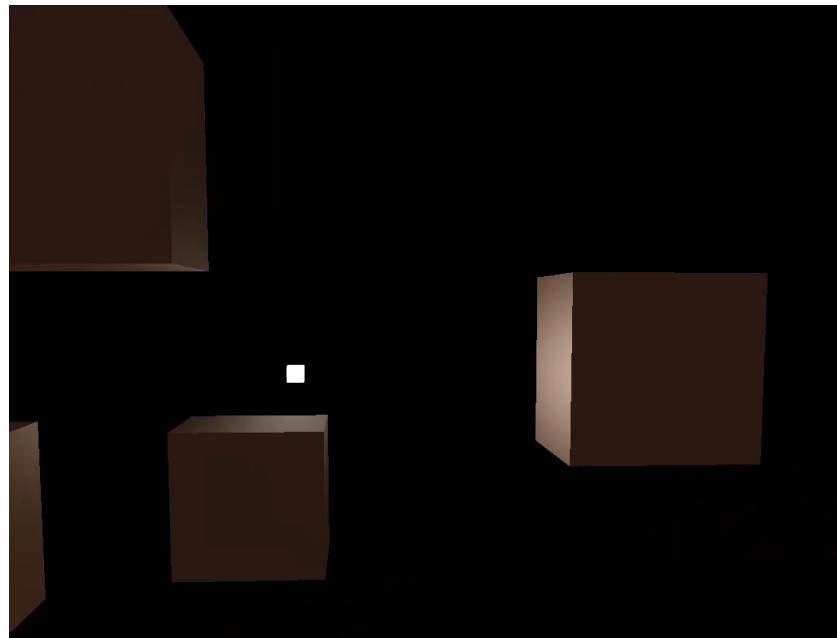
مقدار ۳۲ در کد بالا را مقدار shininess می‌گویند، هر چقدر این مقدار بیشتر باشد درخشش بازتاب در آن نقطه بیشتر می‌شود.

در نهایت با جمع پستن این سه مقدار می‌توانیم 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:



شکل ۹.۱ blinn-phong shading

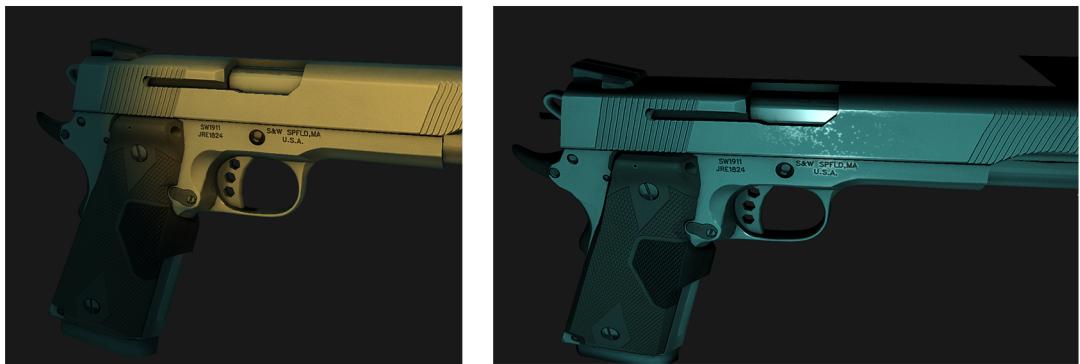
در حال حاضر تمام سطوح مقداری که بازتاب می‌کنند تنها به میزان زاویه با منبع نور بستگی



شکل ۱۰.۱ blinn-phong shading :

دارد، 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 تر از 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:

بعد از استفاده از این specular map برای مدل اسلحه تیجه به شکل زیر در می آید، می بینید که مقدار یازتاب در همه نقاط یکسان نیست:



شكل ١١.١ phong shaded gun :

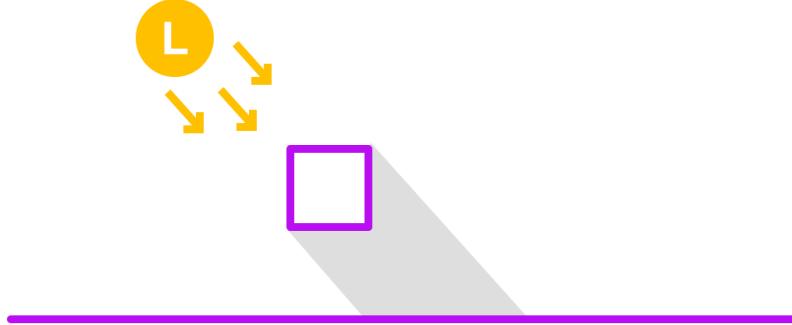


شكل ١٢.١ a gun specular map :

Shadows

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 سرچشم می گیرند، در الگوریتم های ray-tracing به صورت خودکار سایه ها تولید می شوند، دلیل آن نیز نرسیدن پرتو های نور به نقاط است.

روش هایی که برای تولید سایه در real-time استفاده می کنیم شباهت زیادی از لحظه تعریف به روش بالا دارند، باید سطوحی را پیدا کنیم که نور به آن ها نمی رسد، یکی از این روش ها shadow mapping نام دارد، این روش و مشتقات آن در بازی کامپیوترا مورد استفاده قرار می گیرند. shadow mapping روشهای ساده است که در مراحلی که در زیر شرح می دهم قابل تولید است: اساس کار ایجاد نقشه ای است که با استفاده از آن می توانیم مشخص کنیم که نور به چه جسمی زودتر برخورد کرده و چه جسمی به نور نزدیک تر است



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 می آید، در روش shadow mapping ابتدا صحنه را از دید منبع نور به صورت ساده رندر می کنیم، در این عملیات ما fragment shader را خالی می گذاریم، زیرا نیازی به خروجی آن نداریم، تنها کاری که باید انجام شود پر شدن depth buffer است، این کار به صورت خودکار انجام می شود پس ما نیازی به نوشتمن کد در fragment shader نداریم،

shadow depth buffer تشكيل شده را به يك تکسچر متصل می کنيم، به اين صورت shadow map ما آماده است.

بعد از اينكه shadow map آماده شد، حالا صحنه را از اول و با تمام جزئيات موردنیاز رندر می کنيم، و در fragment shader تصميم می گيريم که آيا اين نقطه نزديک ترین نقطه در برخورد با پرتو نور است یا خير، برای اينكار نياز داريم که نقطه اي که در حال پردازش light Space vertex shader و توسط light space هست را به fragment shader ببريم، اين کار در مرحله light space انجام می شود، مقدار اين ضرب به مرحله fragment shader فرستاده می شود و آن جا به شكل زير برای محاسبه نور استفاده می شود:

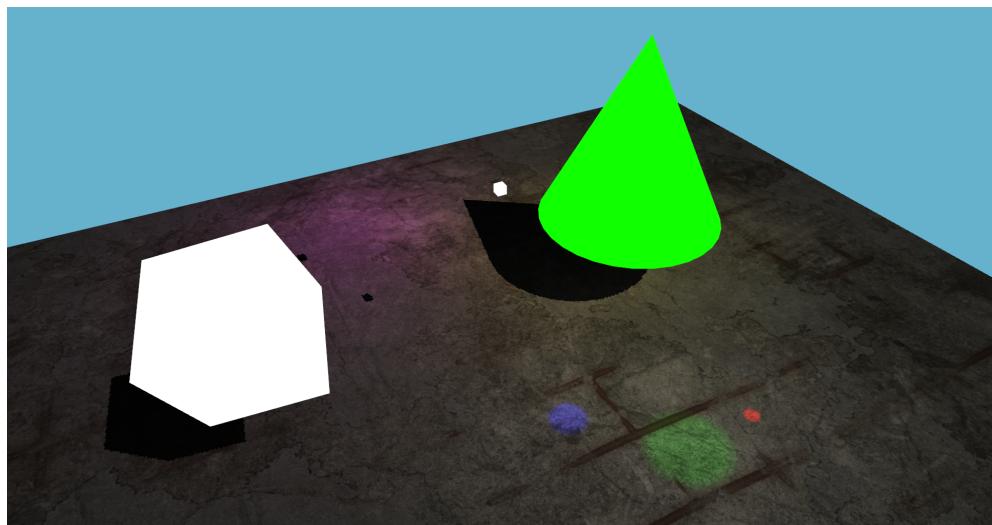
برنامه ۲۶.۱ calculate shadow :

```

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

در تابع بالا ابتدا مقادير $[x, y, z]$ تقسيم بر مقدار w می شوند زيرا light space

به صورت orthogonal matrix محاسبه شده بود، زیرا همانطور که پیش تر گفتیم پرتو های نور با یکدیگر موازی هستند و اگر بخواهیم از دید منبع نور جهان را رندر کنیم باید از view matrix ای 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 کنیم، 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 0.1 به معنی True گردانیم. حاصل کارهای بالا به ساختن سایه منجر می شود:



شکل ۱۴.۱ scene with shadow :

کاربردها

Use Cases

real time Rendering به پروسه تولید یک تصویر از داده ای سه بعدی گفته می شود، rendering اشاره به تولید متنابع 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 استفاده قرار می گیرند.

برای مثال گوشه ای از کاربرد های این نرم افزار ها را در موتور Eevee می بینیم. موتور رندر سه بعدی realtime استفاده شده در نرم افزار blender است، Eevee به منظور افزایش سرعت در زمان توسعه و گاهها 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 real time renderer Eevee پروسه ای بسیار زمان بر و شامل آرمون و خطاهای زیادی می باشد که هزینه هایی زیادی از نظر مالی و نیروی انسانی به جای می گذارد، Eevee به صورت مازولار و منطبق با قرارداد های Blender ساخته شده. همچنین OpenGL با استفاده از Eevee و تکنیک های render استفاده شده در صنعت بازی سازی موفقیت های زیادی کسب کرد.

موتوری که من در این پروژه توسعه دادم از لحاظ نرم افزاری در دسته بندی نرم افزار هایی مانند Shm یا Eevee قرار می گیرد، Eevee مرکز بر تولید سریع و شبیه سازی مطابق با واقعیت تصاویر دارد، اجزا زیادی مانند:

۱. ارائه دو مدل دوربین ortho و perspective

۲. ارائه سه نوع کلی از light caster مانند point ، sun or directional

۳. تولید shadow و highlight ها مانند light effect

۴. کنترل تصاویر، textures و gpu buffer ها

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 specular texture maps و ویژگی های آن ها

۷. ارائه کلاس ها برای shader compile و load

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

... . ۹

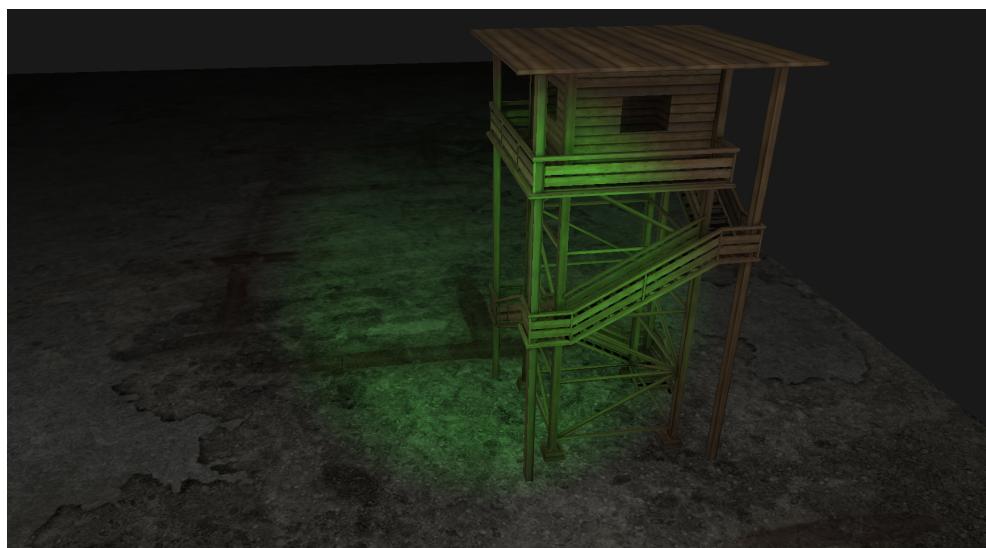
توسط shm ارائه می شود.

3D viewer

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



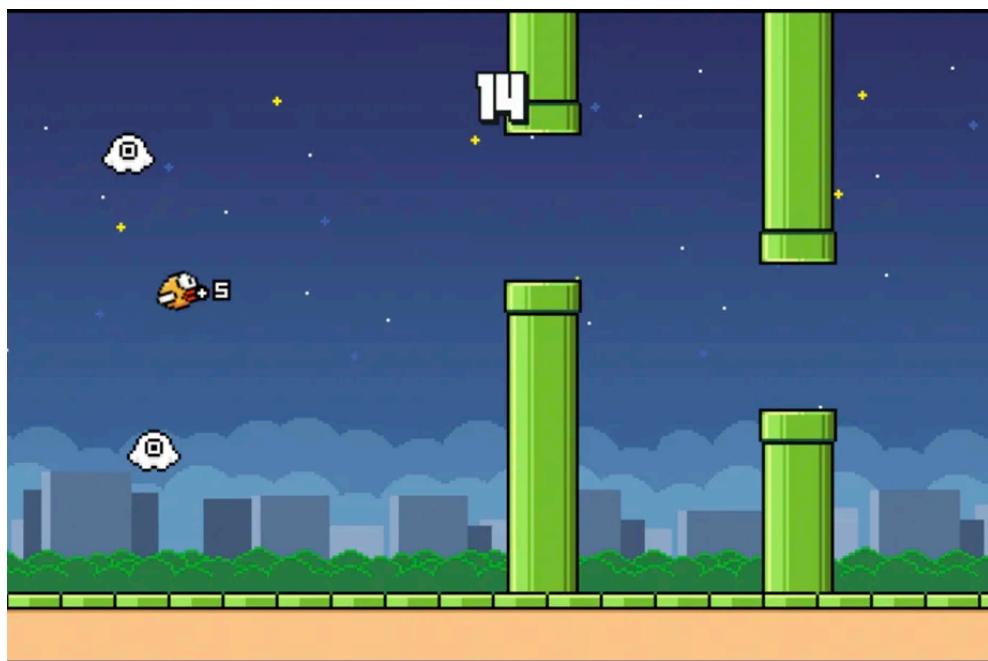
شکل ۱۵.۱ model in microsoft 3d viewer : ۱۵.۱



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

Games

یکی از کاربردهای real-time render engine ها همانطور 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 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:



شکل ۱۷.۱ : original flappy bird

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

برنامه ۱۷.۱ : 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 ۱۸.۱ \square 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 inLoop در main render loop 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 render loop 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 ۱۹.۱ \square loading a 3D model

```

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

```

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

برنامه ۳۰.۱ \square

```

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

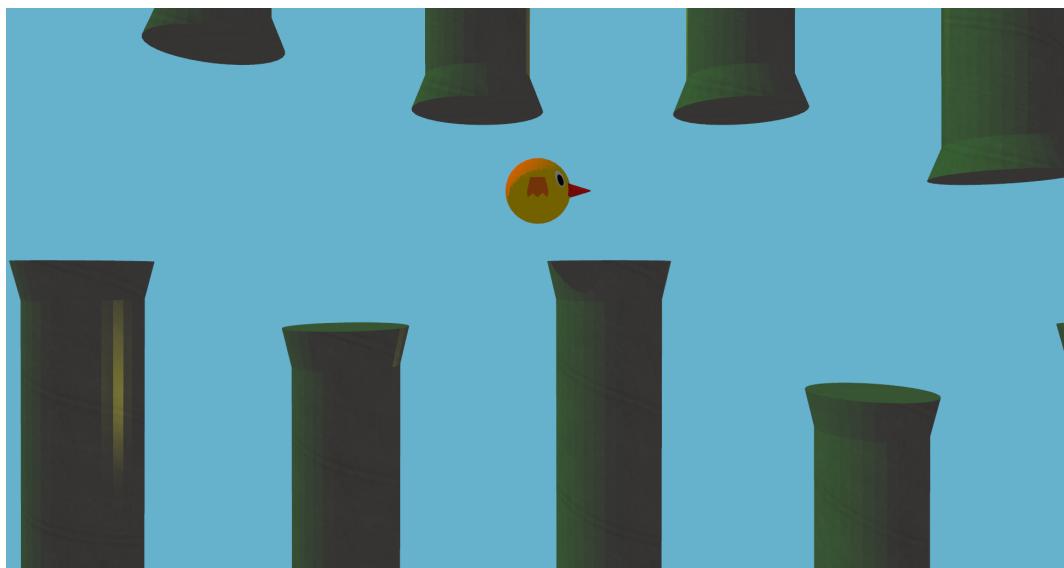
```

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.