# مقدمه ای بر گرافیک کامپیوتر

احمد منصوری و **peter shirley**

December 21, 2009

# 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.

# فهرست مطالب

# 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**, **geometry**, ...) و نمایش آن ها در صفحه را برعهده دارد، در این قسمت ما با استفاده از ریاضیات مدل ها را 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 مستقیما مربوط به یکی از قسمت های موتور در بخش گرافیک است، ابتدای هر فصل فایل های مربوطه به آن فصل ذکر خواهند شد.

فصل ۱

# Renderer

# پنجره ها و کانتکست ها

## openGL

***OpenGL*** یک ***api*** چند زبانه و کراس پلتفرم است که برای به تصویر کشیدن تصاویر دو بعدی و سه بعدی با استفاده از بردار ها 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)* اشتباه گرفت، **OpenGL** به صورت یک interface و یک قراداد انتزاعی در ورژن های مختلف ارائه می شود که فروشندگان و سازندگان*(vendor)* مختلف باید 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 داراست، 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. windows، پنجره ای است که GLFW 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 state و framebuffersها . برای کنترل کردن ورودی ها، glfw از دو روش استفاده می کند، برای ورودی*mouse* از callback function ها استفاده می کند، اما برای ورودی keyboard 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.

شکل ۱.۱: *glfw logo*

# GLAD

کتابخانه ای برای *load* کردن pointer ها به توابع opengl در هنگام runtime. این کتابخانه یکی از کتابخانه هایOpenGL Loading Library است، برای کار با opengl ما حتما باید یکی از این کتابخانه هارا مورداستفاده قرار دهیم تا بتوانیم به توابع opengl 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 پارامتر 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.

برنامهٔ ۱.۱: *load opengl function pointer*

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

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 کامپیوتری جایگاه ویژه ای دارد، مثلث ساده ترین شکلی است که تشکیل سطح میدهد، برای رسم کردن یک مثلث در صفحه نیاز 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، تقاط روی صفحه دوبعدی 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 NDC را به شکل رو به رو نمایش می دهیم:

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

مقادیر $x, y, z$ 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:

برنامهٔ ۲.۱: *points for a triangle*

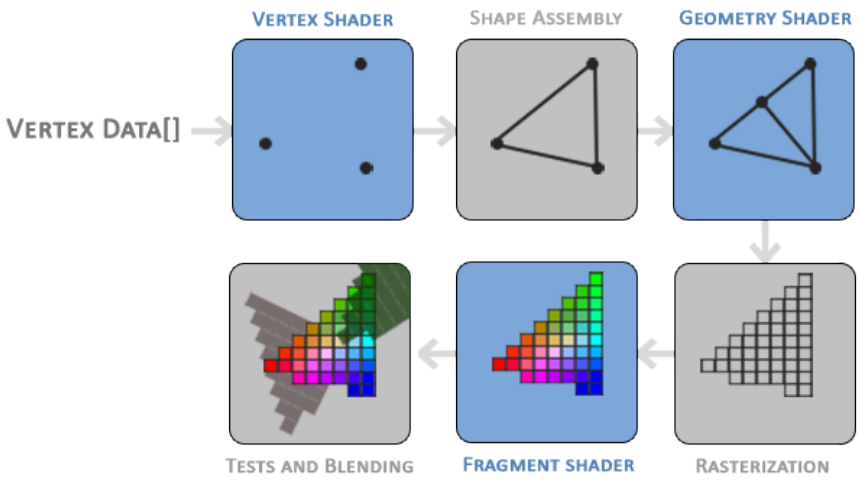
```
float vertices[] = {
    // x   ,  y,     z
    -0.5f, -0.5f, 0.0f, //p1
     0.5f, -0.5f, 0.0f, //p2
     0.0f,  0.5f, 0.0f  //p3
};
```

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 مثلث 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

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 دهند، 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.




شکل ۲.۱: Graphic pipeline: from learnopengl.com

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

برنامهٔ ۳.۱: *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 ساده ترین مدل برای استفاده از vertex shader است، این کد به زبان GLSL(OpenGL Shading Language) نوشته شده است، در خط اول ما نوع ویژگی ها را که پیش تر 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 ۱ مقداردهی کردیم، مقدار چهارم در این فضا برای ما فایده ای ندارد اما در فضای سه بعدی با perspective view به کار ما می آید.

## Upload Data to GPU

# Render Loop

# بخش دوم

# فیزیک

فصل ۲

# Physics

## ۱.۲ Particles

## ۲.۲ Rigid Bodies