Wrapping Low-level Graphics Library for Interactive Application

Chœ Byung-yoon

들어가며

이 글은 게임이나 암튼 그 3D 리얼타임 어플리케이션을 만들 때 꼭 필요하게 되는 부분인 그래픽스 렌더링 엔진 제작에 대해 다룹니다. 저수준 그래픽스 라이브러리를 어떻게 하면 잘 묶어서 좀 사람이 쓸만한 수준으로 만들까 하는 데에 있어 좀 도움을 줄 수 있으면 좋겠다는 생각에서 작성된 글입니다. 모쪼록 이 글을 읽고 그래픽스 렌더링 엔진을 만들 때의 고통이 줄거나, 책상 밑에서 대체 무슨 일이 일어나는지에 대한 이해를 함으로써 암튼 그 3D 리얼타임 어플리케이션 개발에 있어서 도움이 된다면 좋겠습니다. 우리도 인디게임 좀 많이 만들어봐야죠.

** 이 글에서 다루는 저수준 그래픽스 라이브러리는 OpenGL 3.1+을 기준으로 기술됩니다.*

I. 서론

얏호- 모두들 안녕. 하는 건 없는데 고통은 받고 있는 대학원생 최병윤이라고 합니다. 앞서 써놨듯이 이 글은 저수준 그래픽스 라이브러리를 어떻게 하면 좀 사람이 쓸만하게 묶을까에 대한 글입니다 — 좀 더 자세히는 제시 가능한 디자인 초이스들과 왜 그런 선택을 했는지에 대해 설명하는 글입니다. 이 글의 내용 상당수는 KAIST 게임개발동아리 HAJE에서 진행된 프로젝트[1] 개발 중 발생한 렌더링 엔진 개발 경험을 기반으로 합니다. 각 항목들에 대해 얘기하면서 기본적인 설명은 확인을 목적으로 어쨌든 하겠지만 일단은 OpenGL에 대한 기본적인 이해를 하고 있다는 가정 하에 내용이 전개될 거에요. 난 삼각형도 어떻게 띄우는지 모르는데! 하시는 분들은 너무 패닉하지 마시고 다음 URL에서 제공하는 튜토리얼을 진행해 보시면 도움이 될 지도 몰라요. **http://open.gl/**

[1] https://bitbucket.org/haje/mmo

혹시 읽다가 모르겠는 게 있거나 틀린 내용, 제안할 사항이 있다면 주저 없이 저자에게 메일 혹은 트위터로 연락 주세요.

II. 버퍼 다루기

옛날 옛적엔 immediate mode라고 해서 뭐 그릴 때마다 GPU에 데이터를 넘겨주던 시절이 있었죠. 이는 곧 그 비효율성 때문인지 vertex의 정보들을 갖고 있는 buffer들로 대체됩니다. 덕분에 더 빨라지게 된 건 좋은데 어떻게 보면 굉장히 기본적인 동작에조차 좀 더러운 코드를 손에 묻혀야 하게 된 원인이 되었죠. 어떻게 하면 이걸 좀 깔끔하게 감싸넣을 수 있을지 얘기해 봅시다.

A. OpenGL에서의 Buffer Objects

요즘 GPU 아키텍처에는 CPU에서 직접 접근하는 RAM과는 별도의 온–칩 메모리가 달려 있죠. 하도 많은 데이터 액세스가 일어나다 보니 액세스 타임 좀 줄이려고 그렇게 설계를 해 놨는데, 그렇다 보니 여기 메모리도 또 따로 관리를 해 줄 필요가 생겼습니다. OpenGL에선 이걸 라이브러리에서 특정한 몇 개의 목적에 따라 컨트롤할 수 있도록 만들어 놨어요. 이것들을 Buffer Object라고 합니다. 각 buffer object들은 생성generate될 때마다 목적별로 serial한 ID를 부여받게 되고, 얘네를 대충 C/C++ 프로그래밍에서의 포인터와 비슷한 느낌으로 써요. 생성도 했으니 당연히 제거delete도 됩니다. 메모리 관리 열심히 해 줘야 돼요(눈물).

OpenGL은 기본적으로 state machine인데요, 그러다 보니 각 목적별로 buffer를 참조할 일이 있을 때마다 마지막으로 `bind()`한 buffer object를 참조해요. 마지막으로 `bind()`한 buffer object를 상태로서 들고 있는 거죠. 한 예를 들면, 화면에 삼각형들을 그리고 싶다, 그런데 어떤 정점 배열을 참조하지? 할 때 마지막으로 `bind()`된 정점 배열 버퍼의 내용을 화면에 뿌려준다, 이런 식으로 동작한다고 할 수 있어요.

위 내용에 기반해서, buffer object들은 각 종류별로 서로 다른 클래스로 묶어줄 겁니다. 각각은 자신의 ID, 생성, 제거, `bind` 메서드를 공통으로 들고 있을 거고요. Buffer object의 ID가 만들어지는 과정 자체가 API의 생성 콜을 요구하므로 생성 메서드는 `static`으로 해서 그 클래스 자신을 리턴하는 형식을 쓰는 것이 좋습니다Code 1 참조.

```
public class Object
{
  Object(int arg1, int arg2) { /* initialization */ }
  public static Object CreateObject(int arg)
  {
    /* processing */
    return new Object(arg1, arg2);
  }
};
```

CODE 1. 이런 형태의 CreateObject같은 거

B. Vertex Buffer Object(VBO)

VBO는 정점의 속성들을 들고 있는 버퍼입니다. VBO 하나당 속성 하나씩 넣어서 나중에 묶느냐 VBO 하나에 모든 속성 데이터를 다 집어넣느냐는 상당히 취향이 갈리는 문제인 거 같긴 한데, 마지막으로 렌더링 엔진을 만들었을 때는 VBO 하나에 모든 속성 데이터를 넣는 방식을 취했어요. 그 땐 그렇게 해야 물체 하나당 정점 속성 배열 하나가 나와서 VAO(뒤에 다룹니다)의 생성 메서드가 깔끔해진다는 이유였던 것 같은데, 그 디자인 선택이 나중에 **III. 정점 속성**에서 다룰 정점 속성 관련 난제를 만들었습니다. 나중에 한번 더 만든다면 전자를 시도해보고 싶긴 해요.

VBO 생성 메서드를 부를 때 정점 속성들 전부[2]와 이 버퍼를 어떤 용도로 사용할지 `bufferUsageHint`를 인자로 받도록 하면 buffer object를 만드는 데에 충분한 정보를 얻을 수 있습니다. 계속 들고 있어야 하는 건 API에서 부여받은 ID. 파티클 시스템이라도 만든다면 혹시 나중에 정점 정보를 업데이트해야 할지도 모르니 정점 개수와 사용 용도를 들고 있는 것도 좋습니다[3].

C. Element Buffer Object(EBO)

EBO는 정점들을 어떤 순서로 그릴지에 대한 정보를 들고 있는 버퍼입니다. 정점 배열에서의 index들을 순서대로 갖고 있는 배열만 받으면 충분합니다. 이 친구는 ID뿐만 아니라, 나중에 렌더할 때 필요하니 생성할 때 받은 index 배열의 크기도 갖고 있어야 합니다.

---

[2] 자세한 정보는 **III. 정점 속성** 참조

[3] Static인데 업데이트하고 그러면 안 되잖아요.

D. Vertex Array Object(VAO)

VAO는 복수의 VBO와 하나의 EBO를 서로 묶어주는 역할을 하는 오브젝트입니다—만은 우린 많은 정점 속성을 하나의 VBO에 넣고 있기로 했으니까 각각 하나씩만 들고 있으면 충분합니다. 물체를 그리는 데에 필요한 모든 속성(정점들의 속성과 그릴 때의 순서)을 모두 받아 VBO와 EBO의 instance를 생성해 들고 있도록 합시다. VAO 자신의 ID도 물론 들고 있어야겠습니다.

이제 drawable 3D 물체에 자신의 VAO instance를 갖고 있게 하고 그걸 `bind()`한 후 `draw()`하는 것으로 화면에 그 물체를 그리게 할 수 있습니다. 음. 비교적 간단해졌네요(정말?).

III. 정점 속성

OpenGL 감싸줄 때 가장 고민 많이 했던 부분이 여깁니다. `glBufferData`로 정점과 관련된 데이터를 GPU에 업로드해줄 때 데이터 배열 포인터와 사이즈를 던져주면 그 메모리 레이아웃 그대로 올려버리거든요…(멤버 함수같은 게 있다면 그대로 공간이 낭비됩니다). 데이터를 올린 후에 그 데이터에서 어느 부분이 어떤 속성에 대한 정보를 갖고 있는지 역시 `glVertexAttribPointer`로 알려주어야 하는데(이 역할을 하는 API 콜의 연쇄를 setup이라 부르겠습니다), 정점 속성 집합[4]이 서로 다르면 이것도 각각 달라야 하고요.

그래서 렌더링 엔진에서 사용할 정점 속성 집합들의 종류를 제한하고, 이들을 각각 개별의 `struct`로 명시합니다(예를 들자면 정점의 위치와 normal vector, 색깔 데이터를 들고 있는 `VertexPositionNormalColor`따위). 만약 C#처럼 reflection이 지원되는 언어라면 해당 타입에 맞는 setup을 불러주는 helper를 구현할 수 있겠고, 아니라면 각 정점 속성 집합 type마다 데이터 배열과 그에 해당하는 setup을 갖는 `class`를 또 만들어 VBO instance를 생성할 때 그걸 넘겨줄 수 있겠죠Code 2 참조. 그걸 받아서 어떻게 할지는 VBO 측 생성 메서드에서 알아서 처리하도록 하고.

여러 종류의 정점 속성을 사용하다 보면 쉐이더와의 관계 역시 골치아플 수 있는데, 워낙 다양한 쉐이더 코드를 다양한 정점 속성 집합에 대해 공용으로 사용할 일이 많다 보니 그냥 렌더링 엔진에서 사용할 모든 속성들에 대해 일련번호를 붙이고 그걸 다 같이 쓰기로 합의하는 것도 괜찮은 방법인 것 같습니다. World space에서의 위치는 0번, normal vector는

[4] 어떤 정점이 가지는 속성의 집합. 표현하고 싶은 것에 따라 서로 다른 속성들을 들고 있을 수 있으니까요. 어떤 정점은 world space에서의 위치와 색깔, normal vector가 필요하지만 어떤 정점은 screen space에서의 위치와 색깔만 필요하다던가.

```
public class VTypeWrapper : VTypeWrapperBase
{
  VType *data;
  // overrides virtual function of VTypeWrapperBase
  public size_t stride()
  {
    return sizeof(VType);
  }
  // overrides virtual function of VTypeWrapperBase
  public void setup()
  {
    /* setup */
  }
};
```

CODE 2. 대충 이런 느낌의 wrapper

1번, 이런 식으로요. 쉐이더 코드에서도 `layout (location = x)` qualifier를 사용해서 컴파일/링크 전에 속성의 위치를 명시해줄 수 있거든요.

다시 생각하는 거지만 VBO 하나당 하나의 속성만 들어가게 했다면 뭔가 좀 더 깔끔하게 만들 수 있지 않았을까 하는 생각이 자꾸 드네요. 그럼 '정점 속성 집합'같은 개념도 필요 없을 거고, offset은 0이요 stride는 그 속성의 size이니 setup과 관련된 추가적인 레이어도 사라질 테니까요.

IV. 쉐이더 프로그램

정점 데이터를 받아서 화면에 어떻게 뿌려질지를 정하는 건 쉐이더죠. 쉐이더 코드들을 컴파일/링크해서 얻을 수 있는, 프로그램 안의 프로그램을 **쉐이더 프로그램**이라고 합니다. 따라서 모든 쉐이더 프로그램을 만드는 데에 사용되는, 쉐이더 코드 컴파일링과 링킹을 하는 메서드를 갖는 베이스 클래스를 만들어둡시다. 이 베이스 클래스에는 또한 `glUseProgram`을 불러줄 `use()`메서드와 쉐이더 프로그램으로부터 uniform 위치를 얻어올 추상 메서드(쉐이더마다 다를 테니까요), fragment data를 bind할 가상 메서드(동시에 여러 개의 렌더 타겟을 쓸 수도 있으니까요)도 포함됩니다.

이제 서로 다른 목적을 가진 쉐이더 프로그램의 클래스는 이 베이스 클래스를 상속받아 만들어집니다. 자신이 위치를 얻어오고 싶은 uniform을 명시해 주고, 각 uniform에 들어갈 값을 넘겨주는 메서드가 추가됩니다. Uniform에 값을 넘겨줄 때 참조해야 하니까 각 uniform의 위치는 들고 있어야겠죠.

이제 렌더러에서는 어떤 쉐이더 프로그램을 사용해 렌더할 때 이 클래스의 instance를 `use()`하고 uniform에 적절한 값을 넘긴 후에 렌더를 하면 됩니다.

V. Framebuffer와 Texture

Texture와 framebuffer는 마치 종이와 틀 같다고 할 수 있습니다. 만약 당신이 텍스처에 렌더를 하고 싶다면 더욱요.

Texture를 감싸는 클래스는 앞에서 다루었던 buffer object와 크게 다르지 않습니다. 차이가 있다면 texture에 대한 `bind()`인 `glBindTexture`는 같은 texture에 대해서도 `bind target`이 달라질 수 있다는 점, 그리고 생성시의 가능한 동작으로 이미 있는 2D 비트맵 이미지를 가져오는 것 외에 비어 있는 texture로 생성하는 것이 있다는 점 등입니다. 각자 취향과 상황에 맞게 `bind()`와 생성 메서드를 디자인해 주세요.

Framebuffer를 사용하고 싶다면 texture는 자신이 나타내고자 하는 것이 color인지 depth인지를 들고 있는 것이 좋습니다. Framebuffer(이 친구도 buffer object이므로 앞서 거론한 형태에서 크게 벗어나지 않습니다)의 생성 메서드가 그 framebuffer에 붙이고자 하는 texture들의 목록을 받을 텐데, 그 때 color attachment에 붙여야 할지 depth attachment에 붙여야 할지를 알아야 하기 때문이죠.

0번 framebuffer는 미리 예약되어 있는 system framebuffer(네, 모니터 화면이요)이기 때문에, 0번 framebuffer를 `bind`하는 `Unbind()` 메서드를 구현해 두면 추후에 편리합니다.

VI. Drawable

지금까지 열심히 화면에 그리기 위한 도구를 래핑했습니다. 이제 화면에 그려질 수 있는 모든 걸 뭉뚱그린 무언가가 필요하죠! Drawable은 VAO instance와 그걸 `bind()`한 후 `glDrawElements()`하는 `draw()`메서드를 갖고 있는 친구입니다. 기호에 따라 이걸 상속받아 static mesh를 그리는 오브젝트를 만들거나, dynamic particle system을 만들거나 할 수 있어요. 정말 별거 없네요.

하지만 원래 그런 거 같아요. 어떻게 하면 잘 감쌀까에 대한 얘기였으니까, 로우레벨에서 점점 위로 올라가는 순서로 쓰인 이런 형식의 글에선 뒤로 갈수록 쓸 게 없어지는 게 좋은 거 아닐까요? 추상화가 잘 됐단 증거겠죠. 만세!

VII. 렌더러

렌더링 엔진의 마지막 단계입니다. 렌더러는 원하는 렌더 결과를 얻기 위해 다수의 쉐이더 프로그램과 framebuffer를 적절하게 사용합니다. 또한 그려야 할 drawable도 프레임마다 갱신받아야 하죠. 카메라처럼 렌더링 과정 전체를 통틀어 일정하게 함께하는 추가정보 역시 들고 있어도 좋습니다.

먼저 렌더러를 초기화할 때 이 렌더러가 사용할 쉐이더 프로그램과 framebuffer를 초기화해서 들고 있게 합니다. 한 프레임에 여러 번 렌더 콜을 부르지 않게 하기 위해 그릴 drawable을 enqueue하는 메서드를 작성하고, 한 프레임 안에 그려져야 할 모든 drawable이 모였을 때 부를 `render()` 메서드를 작성합니다.

이 `render()` 메서드 안에서는 지금까지 래핑해온 buffer object나 쉐이더 프로그램, framebuffer 등을 자유롭게 활용하여 원하는 렌더 결과가 나올 때까지 iterate해요. 래핑하는 데에 좀 오래 걸리긴 했지만, 이 때 와서 iterate할 때 느끼는 능률 향상은 상당히 뚜렷합니다.

```
void render()
{
  colorFBO.Bind();
  {
    /* FBO init */
    colorShader.Use();
    foreach (var mesh in meshList)
    {
      colorShader.SetUniform
        (
        /* Uniform Setting */
        );
      mesh.Draw();
    }
  }
  colorFBO.Unbind();
  ...
}
```

CODE 3. 이쁘게 감싸면 기분이 조크든요

이제 메인 인터랙티브 어플리케이션에선 이 렌더러와 그려야 할 drawable들을 초기화하고, 매 프레임마다 렌더러에 그려질 drawable들을 전송한 후 `render()`하면 렌더가 진행됩니다.

VIII. 클래스 다이어그램

지금까지의 추상화를 그림으로 그려보면 대략 다음 Figure 1과 같습니다.

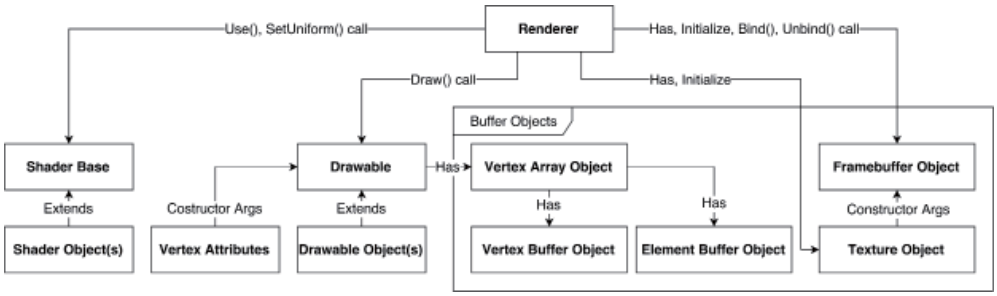

FIGURE 1. 대충 이런 느낌입니다

참고사항

https://bitbucket.org/haje/mmo에서 관련 코드를 열람하실 수 있습니다. `/Client/Rendering`안을 적절히 뒤져보시면 적절히 나옵니다. 읽다가 대체 무슨 얘기를 하는 건가 싶을 때 들어가서 비교분석하면서 읽으시면 도움이 될지도 모릅니다.

Acknowledgements

게임 제작 및 렌더링 엔진 개발 경험을 할 기회를 제공해 준 KAIST 게임제작동아리 HAJE와 MMO 프로젝트 참여진들께 감사드립니다. 특히 프로젝트를 이끌어 준 디렉터 도우진 님과 함께 렌더링 관련 프로그래밍을 한 그래픽스 프로그래머 김의태 님, 래핑과 구조화에 큰 도움을 준 리드 프로그래머 이준희 님께 큰 감사를 드립니다. SSCC 1st