경량 패턴

나무들이 화면을 가득 채운 빽빽한 숲을 볼 때, 그래픽스 프로그래머는 1초에 60번씩 GPU에 전달해야하는 몇백만 폴리곤을 본다.

수천 그루가 넘는 나무마다 각각 수천 폴리곤의 형태로 표현해야한다.
메모리가 충분하다고 해도, 이런 숲을 그리기 위해서는 전체 데이터를 CPU에서 GPU로 버스를 통해 전달해야한다.

나무마다 필요한 데이터는
줄기, 가지, 잎의 형태를 나타내는 폴리곤 메시.
나무 껍질과 잎사귀 텍스처
숲에서의 위치와 방향
각각의 나무가 다르게 보이도록 크기와 음영 값을 조절할 수 있는 매개변수

```
class Tree {
private:
   Mesh mesh;     // 메시
   Texture bark;  // 나무껍질 텍스처
   Texture leaves; // 잎사귀 텍스처
   Vector position;
   double height;
   double thickness;
   Color barTint;
   Color leafTint;
};
```

이를 다 표현하기엔 데이터가 많고, 메시와 텍스처는 크기도 크다.
숲 전체는 1프레임에 GPU로 모두 전달하기엔 양이 너무 많다.

==해결책==

숲은 나무가 수천그루가 있다고 해도, 비슷해보이므로, 모든 나무를 같은 메시와 텍스처로 표현이 가능하다 즉; 나무 객체에 들어있는 데이터 대부분이 인스턴스 별로 다르지 않다.

```
class TreeModel {
private:
   Mesh mesh;
   Texture bark;
   Texture leaves;
};
```

이러한 문제의 해결책으로는 객체를 반으로 쪼개, 모든 나무가 공통적으로 사용하는 데이터를 뽑아내어, 새로운 클래스에 모아보는 것이다.

결론적으로
게임 내에서 같은 메시와 텍스터를 여러번 메모리에 올릴 이유가 전혀 없어,
TreeModel 객체는 하나만 존재하면된다.
이제 각 나무 인스턴스는 공유 객체인 TreeModel을 참조하기만 한다.
Tree 클래스에는 인스턴스별로 다른 상태 값만 남겨둔다.

```
class Tree {
private:
   TreeModel* model;

   Vector position;
   double height;
   double thickness;
   Color barkTint;
   Color leafTint;
};
```

수천 개의 인스턴스
GPU로 보내는 데이터 양을 최소화하기 위해서는 공유 데이터인 TreeModel을
한번만 보낼 수 있어야한다.

그런 후에 나무마다 값이 다른 위치, 색, 크기를 전달하고, 마지막으로 GPU에
전체 나무 인스턴스를 그릴 공유 데이터를 사용하라고 명령하면된다.

요즘 그래픽 카드나 API는 이런 기능을 제공한다.(Direct3D 혹은, OpenGL 모두
인스턴스 렌더링을 지원)

이들 API에서 인스턴스 렌더링을 하려면 데이터 스트림이 두 개 필요하다.
1번째 스트림에는 숲 렌더링 예제의 메시나 텍스처처럼 여러 렌더링 되어야하는 공유 데이터가 들어감.
2번째 스트림에는 인스턴스 목록과 이들 인스턴스를 첫 번째 스트림 데이터를 이용해 그릴 때 각기 다르게 보이기 위한 필요한 매개변수들이 들어간다.

경량 패턴
이름에서 알 수 있듯, 어떤 객체의 개수가 너무 많아서 좀 더 가볍게 만들고 싶을 때 사용.
인스턴스 렌더링에서는 메모리 크기보다 렌더링할 나무 데이터를 하나씩 GPU 버스로 보내는 시간이 중요하나, 기본 개념은 경량 패턴과 같다.

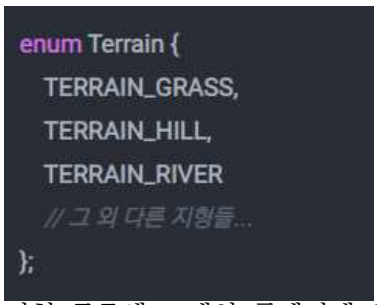
경량 패턴은 객체 데이터를 두종류로 나눈다.
1) 모든 객체의 데이터 값이 같아서 공유할 수 있는 데이터를 모은다. (고유상태 or 자유문맥)
ex) 위의 경우에서는 나무 형태나 텍스처가 이에 해당함.

2) 그 외 나머지 데이터는 인스턴스 별로 값이 다른 외부 상태에 해당한다.
ex) 나무의 위치, 크기, 색 etc

경량패턴은 한 개의 고유상태를 다른 객체에서 공유하게 만들어 메모리 사용량을 줄인다.

** 고유상태(자유 문맥) + 외부상태 로 나눈다. **

지형 정보
나무를 심을 땅도 게임에서 표현해야함.
풀, 흙, 언덕, 호수, 강 같은 지형을 이어 붙여서 땅을 만든다.
여기서 땅을 타일 기반으로 만듦.
땅은 작은 타일들이 모여있는 거대한 격자인 셈이다. 모든 타일은 지형 종류 중 하나로 덮임



지형 종류엔느 게임 플레이에 영향을 주는 여러 속성들이 들어있다.
이들 속성을 지형 타일마다 따로 저장하는 일은 있을 수 없다.

대신 지형 종류에 열거형을 사용한다.

```
class Word {
private:
    Terrain tiles[WIDTH][HEIGHT];
};
```

월드는 지형을 거대한 격자로 관리한다.

```
int World::getMovementCost(int x, int y) {
   switch(tiles[x][y]) {
      case TERRAIN_GRASS: return 1;
      case TERRAIN_HILL : return 3;
      case TERRAIN_RIVER: return 2;
      // 그 외 다른 지형들...
   }
}
```

```
bool World::isWater(int x, int y) {
   switch(tiles[x][y]) {
      case TERRAIN_GRASS : return false;
      case TERRAIN_HILL : return false;
      case TERRAIN_RIVER: return true;
      // 그 외 다른 지형들...
   }
}
```

타일 관련 데이터.
물론 동작에는 문제가 없으나, 이런 식으로 하드코딩되어있는건 좋지 않다.
================================== 데이터를 하나로 합쳐 캡슐화 시키는게 좋음======

아래 계속

-지형 클래스를 따로 만드는게 좋다.-

```
class Terrain {
public:
    Terrain(int moveMentCost, bool isWater, Texture texture)
    : movementCost_(movementCost), isWater_(isWater), texture_(texture) {
    }

    int getMovementCost() const {
        return movementCost_;
    }

    bool isWater() const {
        return isWater_;
    }

    const Texture& getTexture() const {
        return texture_;
    }

private:
    int movementCost_;
    bool isWater;
    Texture texture_;
};

// 모든 메서드를 const로 만든 이유는 Terrain 객체를 여러 곳에서 공유해서 쓰기 때문에, 한
// 메모리를 줄여보겠다고 객체를 공유햇는데 그게 코드에 영향을 미쳐서는 아된다. 이런 이유
```

하지만, 타일마다 Terrain 인스턴스를 하나씩 만드는 비용은 피하고 싶다.
Terrain 클래스에는 타일 위치와 관련된 내용은 전혀없음.

경량 패턴식으로 보면 모든 지형 상태는 고유하다. => 자유 문맥에 해당.

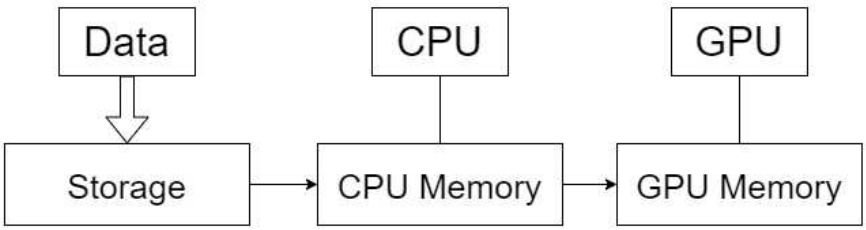
게임 엔진을 사용하는 경우, (유니티 기준)
Inspector에 보면, static 토글 버튼이 있다.
이는 static bathcing이라는 개념의 움직이지 않고 정적 요소들을 하나의 그룹으로 만들어서 drawcall을 줄요주는 역할을 한다.

우리는 대체적으로 하나의 맵을 만들 때, 프랍들을 같이 static 체크하여 하나의 패키징으로 만들어 사용하고 있다.

***드로우 콜이란
CPU는 현재 프레임에 어떤 것을 그려야할지 정해야한다.
GPU에 오브젝트를 그리라고 명령을 호출하는데, 이 명령이 드로우콜이다.

게임의 오브젝트를 화면에 렌더링 하려면, 우선 오브젝트가 렌더링 대상인지 판단한다.
==> 이 과정이 컬링이다.
컬링을 거친 오브젝트가 렌더링 되기 위해선 CPU에서 GPU에 다음의 정보를 줘야함.
1. 메시정보
2. 텍스처 정보
3. 쉐이더 정보
4. 트랜스폼 정보
5. 알파 블렌딩 여부
6. 기타.

메시, 텍스터, 쉐이더 등의 정보는 스토리지에 보관되었다가, CPU가 이를 읽어 들여
CPU 메모리에 데이터를 올린다.

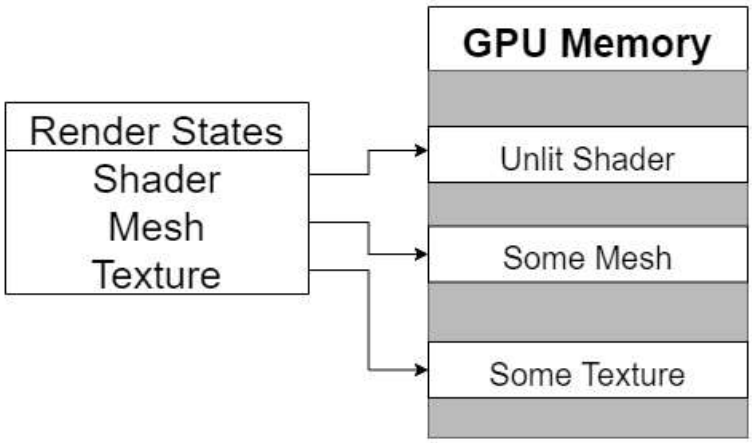


그 후, CPU 메모리에 있는 정보들을 GPU 메모리로 복사한다.
정보들은 GPU 메모리에 있어야 GPU가 사용할 수 있다.

이러한 복사과정이 매 프레임마다 일어난다면, 성능을 많이 잡아먹을 것이다.
따라서 로딩 시점에 데이터를 미리 메모리에 올려둔다.

오브젝트를 렌더링하기 시작하면 GPU에 어떤 텍스처를 사용할지,
어떤 Vertex를 사용할지, 어떤 쉐이더를 사용할지 순차적으로 알려줘야한다.

이런 정보들은 GPU 상태 정보를 담는 테이블에 저장된다.
이 테이블을 렌더 상태라고 부르며, 각 요소는 GPU 메모리를 가리키는 포인터를 저장.



Render States에는 알파 블렌딩 여부, Z테스트 여부, 기타 등등의 정보도 포함됨

Texture, Vertex, Shader 순서로 알려준다. -> GPU 상태 정보를 담는 테이블에 저장.

CPU가 렌더 상태를 변경하는 명령을 GPU에 보내고 나면,
CPU는 GPU에 메시를 그리라는 명령을 보낸다. 명령 = 드로우콜

GPU는 DP Call을 받으면, 렌더 상태의 정보들을 바탕으로 오브젝트 메시를 렌더링함.

한 오브젝트의 메시가 렌더링 되었다면, CPU는 또 다른 오브젝트를 렌더링 하기 위해

Texture, Vertex, Shader의 정보를 변경하는 명령을 한다.

.
그렇다면 이 렌더 상태 정보도 변경이 된다. 그후 DP Call을 받은 GPU가 다시 오브젝트를 렌더링함.

이처럼 한 오브젝트를 그릴 때마다 CPU가 매번 렌더 상태 정보들을 변경하라는 명령 후 DP Call을 해준다.

CPU가 GPU에 명령을 보낼 때, 명령들을 잠시 저장하는 버퍼가 존재하는데,
이를 커맨드 버퍼라고 부른다.
커맨드 버퍼가 존재함으로써, CPU와 GPU는 서로 비동기적으로 일을 처리할 수 있다.
커맨드 버퍼는 그래픽스 API마다 구현 방식이 다르며, 여러개의 커맨드 버퍼나 여러개의 쓰레드를 사용하기도 한다.

문제는 드로우콜에서 사용되는 명령들이 모두 GPU가 알아들을 수 있는 명령들로 변환되어야 함.
=======> 이것이 CPU 오버헤드를 발생시키며 ㄸ라서 드로우 콜은 대게 CPU 병목 현상의 주 원인이다.

병목 현상을 막기 위해서는 드로우콜 횟수 자체를 줄여야 한다.

==> 병목 현상은 엄청난 양의 데이터를 순식간에 내보내어, 메모리가 소화를 제대로 못해서 성능이 뜰어질 수 있다.

드로우콜 발생 조건

기본적으로 오브젝트를 그릴 때, mesh가 1개, material 1개 라면 드로우콜이 한번 일어남.
mesh가 여러개라면 드로우콜도 여러번 일어남.
ex)
한 오브젝트의 메시가 10개라면 해당 오브젝트를 렌더링 하는데, 드로우콜이 10번 발생한다.
만약 이런 메시가 10개인 오브젝트가 20개라면, 드로우콜은 200번 일어남.

그러므로 오브젝트 파츠는 적을 수 록 좋다.

마찬가지로, 메시는 1개인데, 머터리얼이 여러개라면 여러번의 드로우콜이 발생.(서브메시)

쉐이더 내에서 멀티패스로 두 번 이상 렌더링을 하는 경우도, 드로우콜이 여러번 발생한다.
(카툰 렌더링 쉐이더에서 추가적으로 외곽선을 그려주는 경우.

Batch & set Pass
유니티에서는 배치와 셋패스를 구분하는데,
배치는 넓은 의미의 드로우 콜이고, 셋패스는 쉐이더로 인한 렌더링 패스 횟수이다.
오브젝트를 렌더링 하는 중 머터리얼이 바뀌면 쉐이더 및 파라미터들이 바뀌면서
setPass가 증가한다.
이 때 많은 상태 변경들이 일어나기 때문에, setPass도 CPU 성능을 많이 잡아먹는다.

ex) 10개의 오브젝트가 같은 머터리얼을 사용한다면 setPass는 1이다.
그러나, 10개의 오브젝트가 각기 다른 머터리얼을 사용한다면 setPass의 횟수는 10이다.

다른 메시를 사용하더라도 동일한 머터리얼을 사용한다면 setPass는 한번만 일어난다.