# Components

* 컴포넌트 클래스 이름은 Component로 시작.

# ComponentVertex

* glbase.h에 포함됨.
* 디폴트 생성, 쉐이더 프로그램 id를 통한 생성이 가능
* 정점 속성값(pos, color, normal, texture)들과 index(EBO용)의 모음.
* append\_(속성이름)을 통해서 한 정점의 속성을 추가할 수 있음.
* clear\_(속성이름)을 통해서 모든 정점의 해당 속성을 지울 수 있음.
* set\_shader\_program은 쉐이더 프로그램 id를 받아 어떤 쉐이더 프로그램을 사용할지 설정함. (디폴트 생성자를 통해 생성한 경우)
* init은 모든 입력된 속성을 버퍼에 넣음. Init\_(속성이름)으로 특정 속성만 버퍼에 넣을 수도 있음.
* load\_obj는 오브젝트 파일 경로와 면 당 인덱스 개수(삼각형 폴리곤이면 3)을 받아 정점 속성들을 불러옴.
  + load\_obj(“파일 경로”, 3) 후에 init()을 호출하는 것으로 오브젝트를 불러와 프로그램에 적용할 수 있음.

# ComponentRender

* glbase.h에 포함됨.
* 생성자에 ComponentVertex가 필요함.
* ComponentVertex가 들고 있는 정점들을 그릴 수 있도록 도와주는 클래스.
* append()를 통해 아래의 Draw(…)Detail들을 추가할 수 있음.
* clear()를 통해 모든 Draw(…)Detail들을 지울 수 있음.
* draw()를 통해 모든 추가된 Draw(…)Detail들을 참고하여 객체를 그림.
* DrawArrayDetail
  + DrawArray에 쓰일 정보 구조체 {mode, first, cnt}.
  + append( DrawArrayDetail{ GL\_TRIANGLES, 0, size } );와 같이 쓰임
* DrawElementsDetail
  + DrawElements에 쓰일 정보 구조체 {mode, cnt, type, indices}.
  + Append( DrawElementsDetail{ GL\_TRIANGLES, size, GL\_UNSIGNED\_INT, nullptr };와 같이 쓰임
* get\_draw\_desc()를 통해 현재 이 컴포넌트가 어떤 식으로 정점들을 그리는지 설명하는 문자열을 받아올 수 있음.
  + std::cout << get\_draw\_desc() << ‘\n’; 과 같이 쓰임.
* texture 멤버를 수정 가능함.
* texture가 nullptr가 아닐 경우 해당 texture를 적용함.

# Texture

* texture.h에 포함됨.
* 쉐이더 프로그램 id를 통해 생성이 가능.  
  쉐이더 프로그램 id와 텍스처 파일 경로를 통해 생성이 가능.
* load()를 통해 특정 파일 경로의 텍스처를 불러올 수 있음.

# ComponentCoord

* coord.h에 포함됨.
* 디폴트 생성, 시작 위치(glm::vec3 등)를 통한 생성이 가능
* movement, rotation, scale 멤버 변수들이 존재
  + +=, -=, \*=, /=, = 연산자가 오버로드 되어있음.
  + movement += 1.0f; scale \*= 3.0f; 등으로 변환을 기입 가능함.
  + 모든 변환은 movement -> rotation -> scale 순서로 곱해짐.  
    (실적용 순서는 거꾸로인 scale -> rotation -> movement)
* set\_shader\_world\_transform()을 통해 쉐이더에 변환 적용.
* adopt\_base()를 통해 특정 ComponentCoord의 좌표계 위로 올릴 수 있음.  
  (”책상 위에 핸드폰을 올린다.”, “태양계에 지구가 속한다.”와 같이 상대적인 좌표 표현을 위해 도입함. base 좌표계가 움직이면 따라 움직이는, 일종의 플랫폼을 두는 것과 같은 효과임.)
* getpivot(), getradians(), getscales(), getmat()를 통해 변환된 정보를 가져올 수 있음.
  + 예를 들어, 특정 각도 이상 회전을 막고 싶을 때 getradians()를 통해 각도를 확인하면서 회전을 진행하는 것이 가능함. 특수한 상황이 아니라면 이 함수들은 쓸 일이 없음.

# ComponentCollide

* collision.h에 포함됨.
* aabbs, obbs 멤버에 AABB, OBB를 추가하여 바운드 볼륨을 형성 가능함.
* collide()를 통해 다른 ComponentCollide와의 충돌 여부를 검사할 수 있음.
* AABB (Axis-Aligned Bounding Box)
  + AABB.h에 포함됨.
  + 디폴트 생성과 x,y,(z,…)축 최소 범위를 나타내는 벡터 mins와 x,y,(z,…)축 최대 범위를 나타내는 벡터 maxs를 받아 생성하는 것이 가능.
  + mins와 maxs 멤버를 수정하여 바운딩 박스의 범위를 조절 가능
  + collide()를 통해 다른 AABB와의 충돌 여부를 검사할 수 있음.
* OBB (Oriented Bounding Box)
  + OBB.h에 포함됨.
  + 디폴트 생성이 가능.  
    pivot 위치 벡터, x,y,(z…) 방향 절반 길이들을 담는 벡터, x,y,(z…) 축 기준 회전 각도들을 담는 벡터를 받아 생성이 가능.
  + transform()을 통해 pivot 위치를 변경 가능  
    radians와 axis\_half\_len 멤버가 공개되어 있음.
    - 무조건, 충돌 검사를 하기전에  
      radians 멤버에 rotation(ComponetCoord에 있는)을 더하고,  
      axis\_half\_len 멤버에 scale(ComponentCoord에 있는)를 곱해야 함.
  + collide()를 통해 다른 OBB와의 충돌 여부를 검사할 수 있음.

# ComponentPhysic

* physics.h에 포함됨.
* velocity(속도), accel(가속도), velocity\_angle(각속도), accel\_angle(각가속도) 멤버를 수정 가능함.  
  각속도는 실제 선운동에 반영되지 않고, 제자리 회전에만 관여함.
* time\_unit 멤버를 수정 가능함.
* update() 함수를 통해 물리적 이동, 회전을 적용함. 위치 레퍼런스와 각도 레퍼런스, delta\_time을 인자로 받아 받은 위치와 각도를 변환시킴.
  + ComponentCoord인 coord가 있다고 할 때,  
    update(coord.movement, coord.rotation, frame\_time);과 같이 사용 가능함.  
    coord에 들어있는 movement 정보와 rotation 정보를 업데이트하는 것임.
  + time\_unit = (velocity의 시간 단위값) / (delta\_time의 시간 단위값).  
    시간 단위가 ms인 데 속도는 s기준일 경우 time\_unit을 1000으로 설정하면  
    time\_unit = (1s)/(1ms) = 1000으로, 실제 계산 시 단위가 맞춰짐.

# ComponentID

* IDSystem.h에 포함됨.
* 디폴트 생성이 가능함.
* 아이디들을 클래스 단위로 관리함.  
  ComponentID 객체들은 그 어떤 상황에서도 서로 id가 다름이 보장됨.
* getid()를 통해 객체의 id를 알아낼 수 있음.

# ID\_Inst\_Pair

* IDSystem.h에 포함됨.
* ComponentID와 특정한 타입 Ty의 짝으로, Ty 인자를 받아 생성됨.

# Camera

* camera.h에 포함됨.
* 디폴트 생성이 가능.  
  fovy, aspect, znear, zfar를 통한 생성이 가능.  
  left, right, bottom, up, znear, zfar를 통한 생성이 가능.
* ortho()와 perspective()를 통해 각각 직각투영모드/원근투영모드를 설정할 수 있음.
* set\_shader\_camera\_transform()을 통해 쉐이더에 뷰 변환, 투영 변환을 적용.
* eye, at, up에 대해 CoordComponent와 PhysicComponent 클래스인 멤버를 가지고 있음.  
  coord\_component\_(eye/at/up), physic\_component\_(eye/at/up)을 수정 가능함.
* 원근투영과 직각투영에 쓰이는 정보인 perspective\_attribute와 ortho\_attribute 멤버가 수정 가능함.
  + perspective\_attribute엔 fovy, aspect, znear, zfar 정보를 담음.
  + ortho\_attribute엔 left, right, bottom, up, znear, zfar 정보를 담음.

# Shader

* shader.h에 포함됨.
* enum class Type을 이용할 수 있음.  
  Type에는 VERTEX\_SHADER, FRAGMENT\_SHADER가 들어있음.
* 소스 문자열과 Type을 받아 생성 가능.
* id()를 통해 할당된 쉐이더 id를 얻어올 수 있음.

# ShaderProgram

* shader.h에 포함됨.
* Shader 객체 여러 개(가변 인자)를 받아 생성 가능.  
  생성자의 호출이 곧 쉐이더들을 링크하는 과정임.
* use()를 통해 쉐이더 프로그램을 활성화할 수 있음.
* id()를 통해 할당된 쉐이더 프로그램 id를 얻어올 수 있음.

# Timer

* timer.h에 포함됨.
* 디폴트 생성이 가능. 실행할 함수 포인터와 fps를 받아 생성이 가능.
* component\_id가 이용가능함.
* setFPS()를 통해 fps 설정이 가능함.
* setfunc()를 통해 함수 설정이 가능함.
* run()을 통해 타이머를 작동/재작동 시킬 수 있음.
* stop()을 통해 타이머를 일시 정지할 수 있음.

# Sound

* sound.h에 포함됨.
* Sound::make()를 통해서만 생성이 가능함.  
  enum class인 mode를 통해 반복 재생할지, 한 번 재생할지 결정.  
  file\_path, mode, volume(기본값 1.0), gradient(기본값 0.5)가 필요함.
  + 동시 사운드 재생시, 나중에 재생된 사운드는  
    volume \* gradient^(현재 재생되고 있는 사운드 수 – 1)의 크기로 재생됨.
* enum class sound\_tag를 이용할 수 있음.  
  사운드에 태그를 붙여 특정 태그의 사운드를 일괄적으로 처리하는 데 쓰임.
* play(), pause(), stop()은 이름대로의 역할을 함.
* mute(), listen()은 각각 음소거/음소거 해제의 효과가 있음.
* amplify()는 볼륨에 주어진 값을 곱함. 증폭/감쇠
* play()부터 amplify()까지의 모든 함수는 앞에 tag\_를 붙이면 특정 태그가 붙은 사운드에 대한 일괄 처리를 하는 함수가 됨.
* tag\_push()는 특정 태그로 사운드를 등록하고, tag\_clear()은 특정 태그의 사운드를 모두 소멸시킴. clear()은 모든 사운드를 소멸시킴.
* 종료 처리 후 채널 정비, 반복 재생을 위해서 주기적으로update()를 호출해줘야 함.
* (게임의 한 프레임에 사운드 업데이트를 포함하는 것이 좋음.)

# EventHandler

* event\_handler.h에 포함됨.
* 디폴트 생성이 가능함.
* component\_id가 이용 가능함.
* 다양한 이벤트 처리를 위한 인터페이스 클래스.  
  실제 적용할 이벤트 처리자는 이 클래스를 상속받아 구현해야 함.
* add\_event()를 통해 이벤트 큐에 이벤트를 삽입하거나  
  다른 이벤트 큐에 id를 통해 접근해 이벤트를 삽입할 수 있음.
* run()을 오버라이드하여 이벤트 처리 동작을 구현해야 함.
* DefaultEventHandler
  + EventHandler가 무의미한 경우, 이 클래스의 객체를 통하여 해결이 가능함.
  + run()을 실행하면, 이벤트 큐를 clear()하는 일 밖에 하지 않음.

# Scene

* scene.h에 포함됨.
* EventHandler를 받아서 생성해야 함.
* 다양한 씬을 만들기 위한 인터페이스 클래스.  
  실제 적용할 씬은 이 클래스를 상속받아 구현해야 함.
* update()와 render()를 오버라이드하여 프레임 업데이트를 구현해야 함.
* event\_handler는 자식 클래스에서 이용이 가능함.

# GameObject

* gameobject.h에 포함됨.
* EventHandler를 받아서 생성해야 함.
* component\_id가 이용 가능함.
* 다양한 게임 오브젝트들을 만들기 위한 인터페이스 클래스.  
  실제 게임 오브젝트들은 이 클래스를 상속받아 구현해야 함.
* update()와 render()를 오버라이드하여 프레임 업데이트를 구현해야 함.
* component\_vertices, component\_renders, component\_coords, component\_physics,  
  event\_handler는 자식 클래스에서 이용이 가능함.

# GameIDObject

* gameobject.h에 포함됨.
* EventHandler를 받아서 생성해야 함.
* component\_id가 이용 가능함.
* 다양한 게임 오브젝트들을 만들기 위한 인터페이스 클래스.  
  실제 게임 오브젝트들은 이 클래스를 상속받아 구현해야 함.
* update()와 render()를 오버라이드하여 프레임 업데이트를 구현해야 함.
* component\_vertices, component\_renders, component\_coords, component\_physics,  
  event\_handler는 자식 클래스에서 이용이 가능함.
* 모든 컴포넌트들은 component\_id로 치환되어 있음.
* id를 통해 실제 컴포넌트에 접근할 수 있다는 전제 조건으로 설계됨. (예를 들어 서버가 존재하는 경우)

# Server

* server.h에 포함됨.
* vertice\_components, render\_components, coord\_components, physic\_components를  
  ID를 key로 하는 해쉬 테이블을 통해 관리함.
* 각 해쉬 테이블들은 공개되어 있음.

# BehaviorTree

* behavior\_tree.h에 포함됨.
* 루트 노드를 받아 생성.
* enum class Result를 이용할 수 있음.  
  노드의 세 가지 상태 FAIL, RUNNING, SUCCESS을 표현함.
* Node
  + Leaf, Selector, Sequence 노드의 부모  
    add\_child(), add\_children()을 통해 자식을 연결 가능함.  
    사용자가 노드들을 만들어 놓고 연결한 뒤  
    루트 노드를 통해 BehaviorTree를 생성하는 것임.
* Leaf
  + 실제 동작을 하는 노드
* Selector
  + 자식 노드 중 한 가지를 선택해 실행하는 노드. (우선순위: 좌->우)
* Sequence
  + 자식 노드들을 순서대로 실행하는 노드. (실행순서: 좌->우)

# Game

* game.h에 포함됨.
* Argc, argv, 스크린x, 스크린y, 스크린width, 스크린height, 게임이름  
  을 통해 생성함.
* 다양한 게임을 구현하기 위한 인터페이스 클래스.  
  실제 게임은 이 클래스를 상속하여 구현해야 함.
* run()을 통해 게임을 실행시킬 수 있음.