CSED451 Assignment #3-1

3D DRAWING REPORT

Team: GPS

CSE 20202405 문민재 csmmj4594

CSE 20202728 김진수 fusion4268

**Overview of Program**

본 프로그램은 과제 2D Drawing에서 플레이어와 도둑이 움직이고 포즈를 변환하는 과정을 animation으로 구현하여 더욱 생동감 있게 구현하였다. 추가적으로 구현된 부분은 1) 플레이어와 도둑을 단순 도형이 아닌 인간형 캐릭터로 형상화, 2) 추격을 진행할 때 움직이는 것을 표현하기 위한 하체 animation, 3) 4가지 포즈와 포즈를 변환할 때 발생하는 상체 animation, 4) 점프로 나눌 수 있다.

오브젝트 간 계층구조를 활용하여 캐릭터를 인간형 캐릭터를 구현하였으며, 상위 계층의 개체가 회전하면 하위 계층의 개체 또한 같은 각도만큼 회전하게 된다. 선형 보간으로 animation을 구현하였으며, 하체 animation은 일정 주기에 따라 반복적으로 실행되도록 상체 animation은 포즈가 변화할 때만 실행되도록 구현하였다.

기존의 프로그램에서는 도둑이 벽을 통과할 때 할당된 색 (빨강, 초록, 노랑, 파랑)과 플레이어가 벽을 통과할 때 할당된 색이 일치할 때 pass로 처리하였다. 본 프로그램에서는 4가지 색에 따른 포즈를 mapping하여 캐릭터가 벽을 통과할 때 4가지 색에 따른 4가지 포즈를 갖도록 하였다.

점프 기능은 플레이어가 벽의 색과 무관하게 벽을 통과할 수 있도록 한다. 도둑은 벽과의 거리가 일정 거리에 도달하였을 때 일정 확률에 따라 벽을 점프할지 판단하며, 플레이어는 ‘space’키를 이용해 언제든지 점프할 수 있다. 플레이어나 도둑이 점프를 수행할 때 일정 거리만큼 왼쪽으로 움직이게 된다.

**Programming Environment**

Visual Studio 2019

OpenGL – 2.1

Freeglut-MSVC-3.0.0

Glew-2.1.0

Glm-0.9.9.7

**Design and Implementation**

1. 자료구조 설계
2. Color (colors.h)

게임 상에서 사용할 색상 코드와 팔레트 변경을 담당한다.

1. Object (object.h)

맵 상의 오브젝트를 정의하는 클래스이다.  
멤버 변수로 type, color, x, y, z를 가지고 있다. 이들은 각각 오브젝트 타입(0일 경우 wall, 1일 경우 character), 오브젝트의 색깔, 오브젝트를 그릴 때 필요한 x, y, z좌표에 해당한다.

1. Wall: public object (Wall.h)

Object를 상속받아 맵 상의 벽을 정의하는 클래스이다.  
Wall 오브젝트들은 공통적으로 type이 0이며, x, y, z는 벽의 위치를 의미한다. 추가적으로 width, height, depth라는 멤버 변수를 가지는데, 이는 각각 직사각형의 x, y, z축 방향 길이를 의미한다.

1. Character: public object (character.h)

Object를 상속받아 맵 상의 캐릭터 오브젝트(플레이어, 도둑)를 정의하는 클래스이다.  
character 오브젝트들은 type이 1에 해당하며, x, y, z는 torso의 중심 좌표를 나타낸다. 그리고 원의 반지름을 나타내기 위한 변수인 rad와 벽과의 충돌 여부를 나타내는 변수인 isCollided가 있다. 그 외에 인간형 캐릭터 구현에 필요한 여러 멤버변수들을 가지고 있다.

1. treeNode (treeNode.h)

인간형 캐릭터의 hierarchy를 표현하기 위해 필요한 Left Child Right Sibling(LCRS) tree를 구현하는 클래스이다.

현재 modelView matrix의 오른쪽에 곱해질 기본적으로 matrix인 mtx,  
기본 mtx transform에 추가적으로 행해질 transform을 나타내는 additionalTransform,  
이 node에 해당하는 신체 부위를 그려주기 위한 함수 포인터인 draw,  
그리고 sibling과 child 정보를 가지고 있다.

1. pose (pose.h)

플레이어와 도둑의 포즈를 정의한 클래스이다.  
캐릭터의 포즈에 따른 변화는 상체에서만 이루어지므로, 왼팔과 오른팔 각각 2가지씩 총 4가지 각도와 포즈에 따른 색을 변수로 갖고있다.

1. 기본 기능 구현(이전 ASSN 관련)
2. 벽의 생성 및 이동

벽의 생성과 이동은 매 frame(1/60초)마다 호출되는 moveWall() 함수에서 이루어진다. 벽의 형태가 2D에서 3D로 변화함에 따라 벽의 색만 변경하던 것에서 색과 모양을 동시에 바꿔주거나 재생성 위치가 3D 좌표로 달라진 것 외에는 assn2와 동일하다.

1. 충돌 판정

Assn2와 동일하게 진행되었다.

1. 플레이어 캐릭터의 조작

Assn2와 동일하게 구현되었다.

1. 치트 기능

Assn2와 동일하게 구현되었다.

1. Pass/Fail 시의 action
   1. Pass

2D에서 3D로 시점이 변경됨에 따라 카메라의 줌인 및 남은 life의 출력 위치 고정을 여기에서 담당하지 않게 되었다. 그 이외에 기본적인 pass 시 행동(Pass 체크 로직, 벽 속도 증가, 도둑의 포즈 전환 간격 감소, 캐릭터의 전진 등)은 assn2와 동일하다.

* 1. Fail

Assn2와 동일하게 구현되었다.

1. Jump 구현

Assn2와 동일하게 구현되었다.

1. 3D 캐릭터 구현

2D에서 3D로 변경됨에 따라  
head: 원->구, torso: 직사각형 -> Cylinder, limb: 원+직사각형 -> 구+cylinder  
로 변경되고, 2D좌표를 3D좌표로 변경한 것 이외에는 인간형 캐릭터 구현에 필요한 아이디어(LCRS tree 및 hierarchy 구조, pose, 상/하체 애니메이션 등)과 실제 구현은 assn2와 동일하다.

그 외에도, 기존 assn2에서 2D에서 캐릭터를 그려내야 했기 때문에 상체는 정면을, 하체는 측면을 그려내는 이집트 벽화 같은 느낌으로 그려냈지만, 이번 assn3에서는 3D로 캐릭터를 그려내기에 실제 사람과 같은 모습으로 그려냈다.

1. 3D 벽 구현

기존 assn1,2에서 벽 오브젝트를 구현할 때 사용했던 ‘rect’ 객체 대신 벽을 위한 새로운 클래스인 ‘wall’을 정의하였다.

이번 과제에서 도둑이 벽에 충돌하면 벽의 색과 더불어 모양도 바뀌어야 하는데, Cone, Torus, Reversed T-shape, T-shape, 그리고 기본 직육면체 벽 모양의 총 5가지 모양으로 이를 구현하였다.

Cone 모양과 Torus 모양은 glut 내에 정의되어 있는 glutSolidCone 함수와 glutSolidTorus 함수를 사용했고, Reversed T-shape, T-shape, 그리고 기본 직육면체 모양은 drawCuboid 라는 함수를 직접 정의하여 구현하였다. drawCuboid 함수는 중심의 x, y, z좌표와 width, height, depth를 받아 화면상에 cuboid를 그려주는 함수이다.

도둑과 부딪히기 전에는 회색의 기본 cuboid 모양의 벽이었다가, 도둑과 부딪히면 도둑의 색/포즈에 따라 벽의 색/모양이 함께 변경된다. Red – Cone, Green – Torus, Blue – Reversed T, Yellow – T로 매핑된다.

1. 3D 시점 구현

3D 시점의 구현은 display 함수인 display3D() 내에서 이루어진다.

먼저, Matrix mode를 GL\_PROJECTION으로 변경한 뒤, Identity로 초기화 시켜주고, gluPerspective 함수로 perspective projection의 3D 시점을 구현하였다. 이 때, gluPerspective에 필요한 화면 종횡비는 reshape 함수의 argument인 width와 height를 통해 얻어온다.

그리고, Matrix mode를 GL\_MODELVIEW로 바꾼 후, identity로 초기화 시켜주고 바로 gluLookAt 함수를 호출하여 카메라의 위치, 시야 방향, view up vector를 설정해 담고 싶은 화면 각도를 담아낸다.

그 이후에 화면상 그려지는 오브젝트들(wall, player, thief 등)을 화면 상에 그린다.

1. 시점 전환 및 줌 인/아웃
2. 시점 전환

이번 assn3-1에서는 총 3가지의 시점을 구현하였다.  
첫째, 카메라가 플레이어 눈 앞에 있는 1인칭 시점  
둘째, 카메라가 플레이어 오른쪽 어깨 뒤 어딘가에 있는 3인칭 시점  
셋째, XY평면을 비추어 assn2처럼 보이게 하는 3인칭 시점

이러한 세 개의 시점의 전환은 키보드 숫자 키로 이루어지며, 키보드 ‘1’을 입력하면 1인칭, ‘3’을 입력하면 3인칭 시점, ‘9’를 입력하면 XY평면을 보는 시점으로 전환된다.

시점 전환의 구현은 이 3가지 시점에 해당하는 카메라 정보(eye, lookat, up vector)들을 미리 정의해 두고, keyboard function에서 누르는 숫자에 따라 cameraMode라는 변수를 변경하게 했다. 매 프레임마다 호출되는 함수인 frameAction에서 cameraMode에 따라 알맞은 카메라 정보를 가져오고, 마지막으로 이 카메라 정보를 display() 함수 내의 gluLookAt에 전달하는 방식으로 시점 전환을 구현하였다.

1. 줌 인/아웃

Assn1, 2와 동일하게, 캐릭터가 벽을 점프하지 않고 성공적으로 통과할 때마다 화면이 줌되어 더 좁은 영역을 비춰야 한다.

Assn1, 2에서는 gluOrtho2D 함수의 시작과 끝 x, y좌표를 변경하여 줌인을 구현하였다면, 이번 assn3-1에서는 display 및 reshape 함수에서 호출되는 gluPerspective 함수에서의 fovy(시야각) 값을 변경하여 줌인을 구현하였다. Fovy 값을 조절하여 더 좁은 시야각을 가지게 한다면, 3인칭 시점에서 플레이어 캐릭터의 이동에 따라 카메라의 위치를 이동시키지 않아도 마치 카메라가 줌 인 되는 듯한 화면을 구성할 수 있다.

플레이어가 점프 없이 벽을 성공적으로 통과한다면, fovy를 기존의 0.8배로 설정하고, 자연스러운 줌을 위해 zoomFrame 기간동안 천천히 줌 될 수 있도록 fovyPerFrame을 설정하였다.

도둑이 점프하지 않았는데 플레이어가 점프하여 벽을 통과할 경우, 플레이어 캐릭터만 뒤로 이동하므로 플레이어가 화면에서 사라지는 것을 방지하기 위해 fovy를 기존의 1.05배로 설정하여 world상의 더 넓은 영억을 비출 수 있게 설정하였다.

1. 화면 상 고정된 위치에 life 출력(추가 기능)

ㅁㄴㅇㄹ

**How to Run**

프로젝트 폴더의 \bin\x64 폴더로 이동하여 graphics\_assn3.exe를 실행시킨다.

캐릭터의 포즈 변경은 방향키로 입력하며, 위, 아래, 왼쪽, 오른쪽 방향키는 각각 플레이어 캐릭터를 적색, 녹색, 청색, 황색으로 변경시키고, 플레이어 캐릭터는 각 색에 대응하는 포즈를 취한다. 스페이스바를 입력하여 캐릭터를 점프시킬 수 있다.

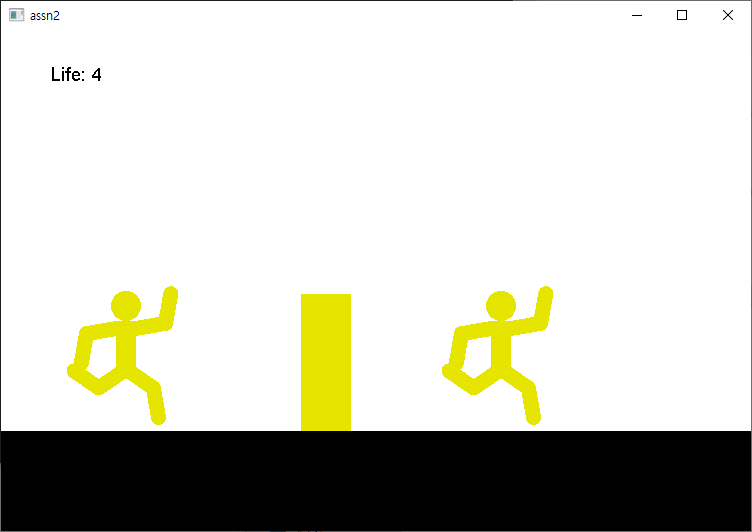
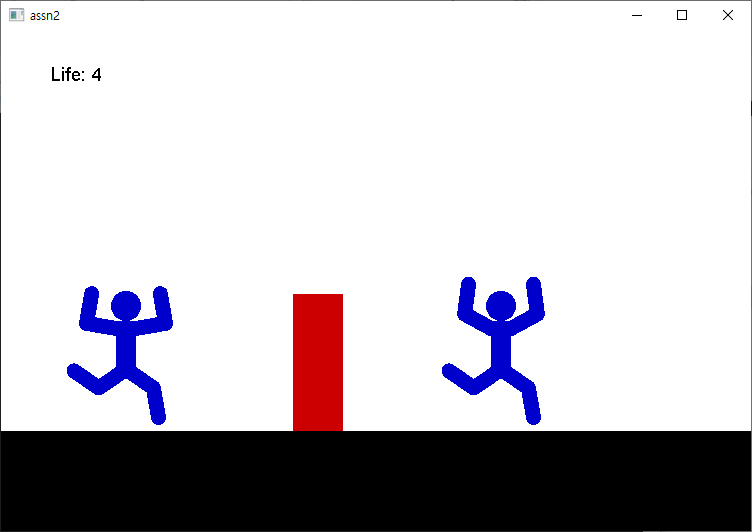
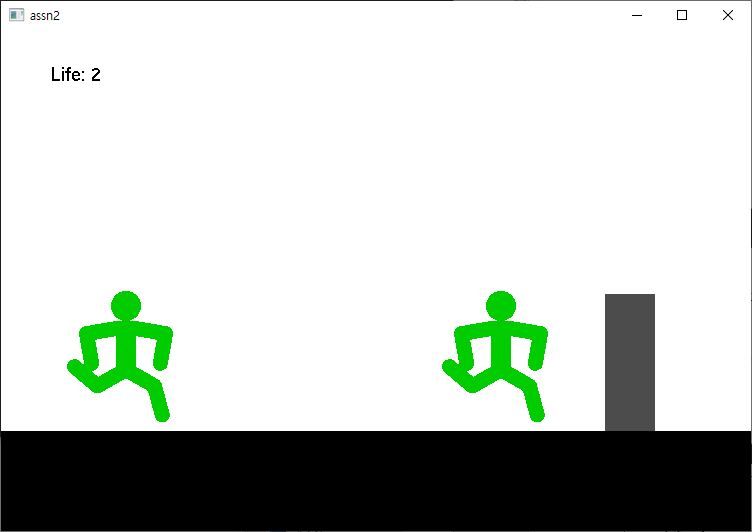
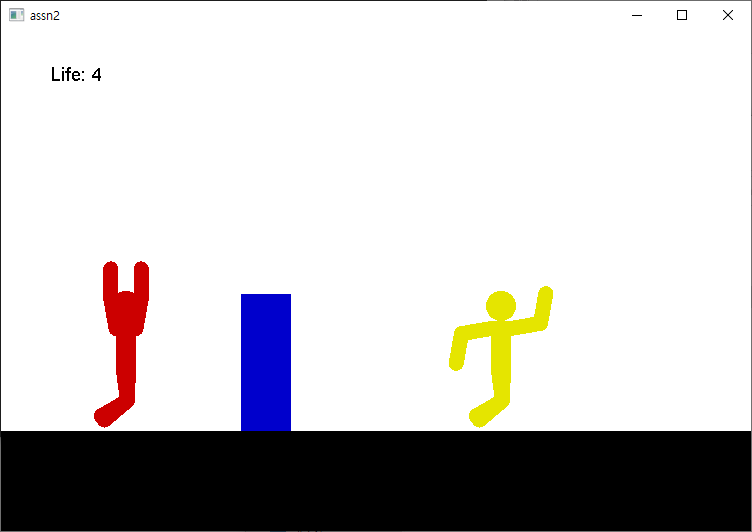
숫자키 1번을 눌러 1인칭 시점으로, 3을 눌러 3인칭 시점으로, 9를 눌러 XY 평면을 바라보는 시점으로 시점을 전환시킬 수 있다.

**Example**

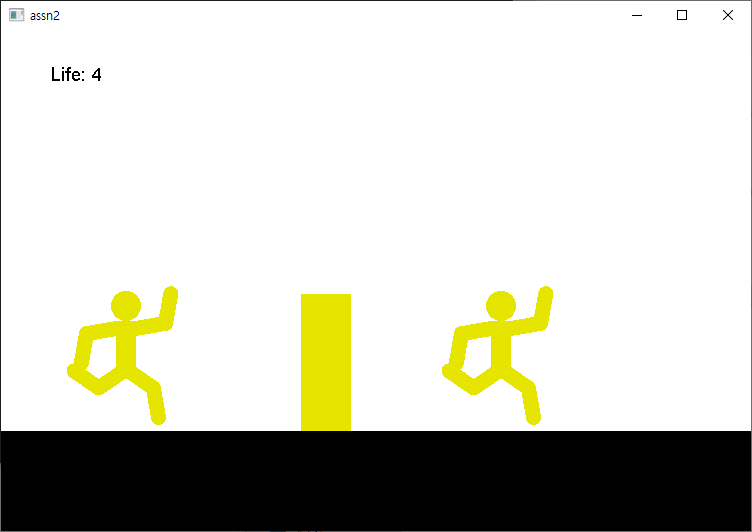
기존 Assn2와 중복되는 부분은 제외하였다.

프로그램 실행 시 화면으로, 좌측 상단에 현재 life가 표시되고, 플레이어는 화면 왼쪽, 도둑은 화면 오른쪽에 위치한다. 벽은 화면 오른쪽에서부터 생성되어 왼쪽으로 이동한다.

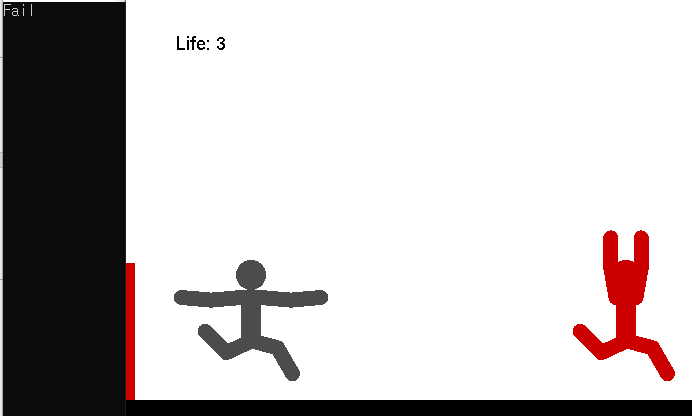
플레이어와 도둑 캐릭터 모두 부드럽게 달리는 모양의 하체 애니메이션 루프를 통해 사람이 달리는 듯한 모습을 그려낸다.



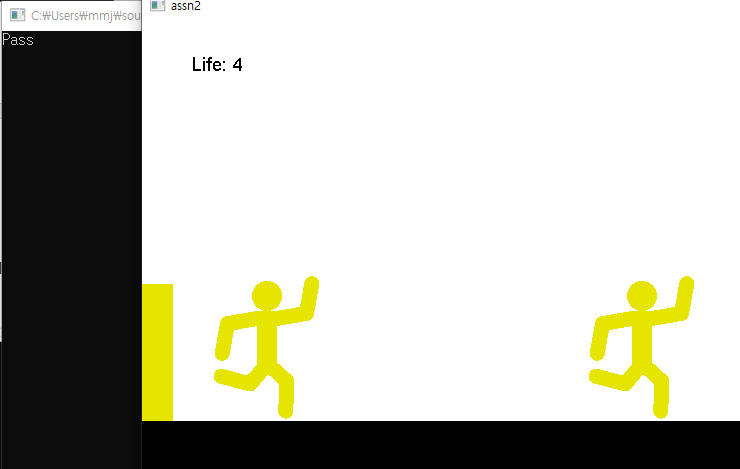
위(적), 아래(녹), 왼쪽(청), 오른쪽(황) 방향키를 눌러 플레이어의 색과 자세를 변경할 수 있다. 이 때, 상체 자세의 관절 전환은 부드럽게 처리된다.



도둑의 색과 자세는 일정시간마다 랜덤하게 변경되며, 도둑 색의 변경 간격은 콘솔에 Pass가 출력되는 횟수가 많아질수록 줄어든다.  
도둑과 벽이 충돌할 경우 벽의 색이 도둑의 색으로 변경되어 플레이어에게 다가온다.

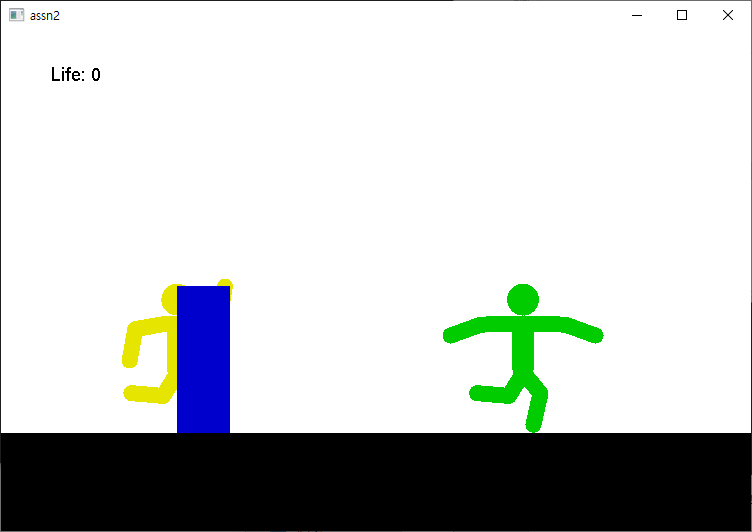
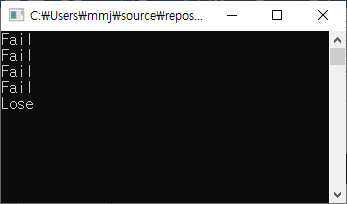


벽이 플레이어를 통과할 때 벽의 색과 플레이어의 색이 다를 경우, Life가 1 줄어들고, 콘솔에는 Fail이 출력된다.

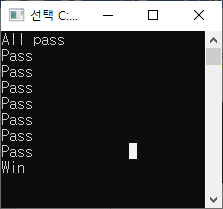


벽이 플레이어를 통과할 때 벽의 색과 플레이어의 색이 같을 경우, 콘솔에 ‘Pass’가 출력되고, 플레이어와 도둑의 거리가 가까워짐과 동시에 화면이 줌인되어 더 좁은 영역을 그린다.

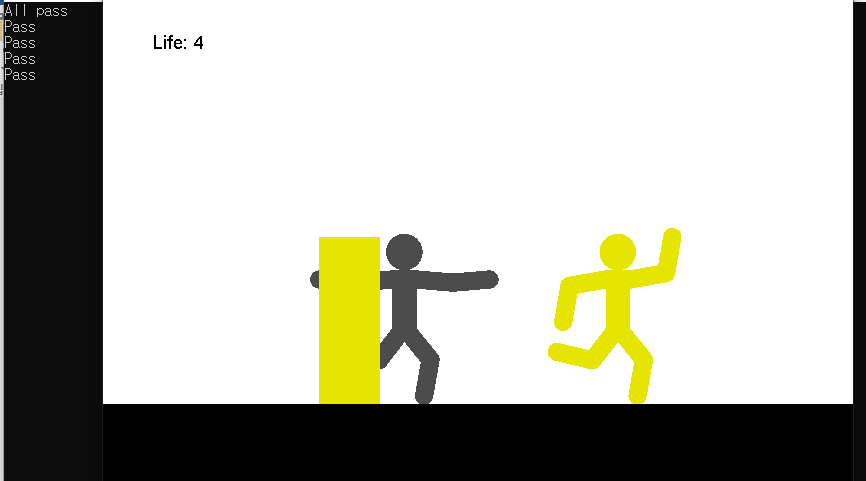
그리고 이후에 생성되는 벽의 속도와 재생성 간격이 빨라진다.



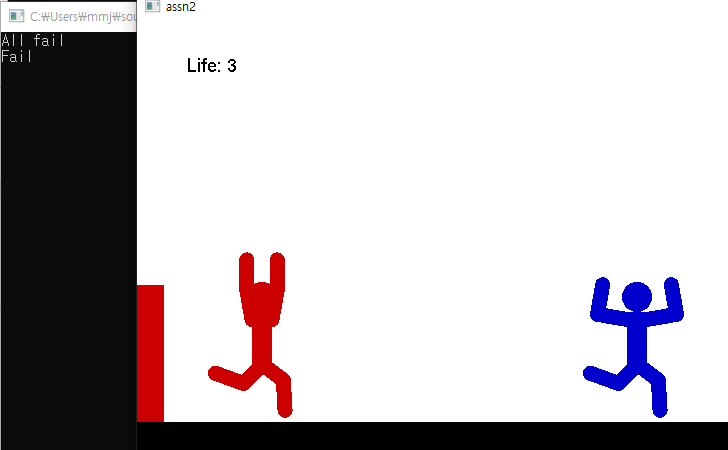
Life를 모두 잃어 0이 되면, 콘솔에는 Lose가 출력되고, 화면이 멈추어 추격이 종료된다.



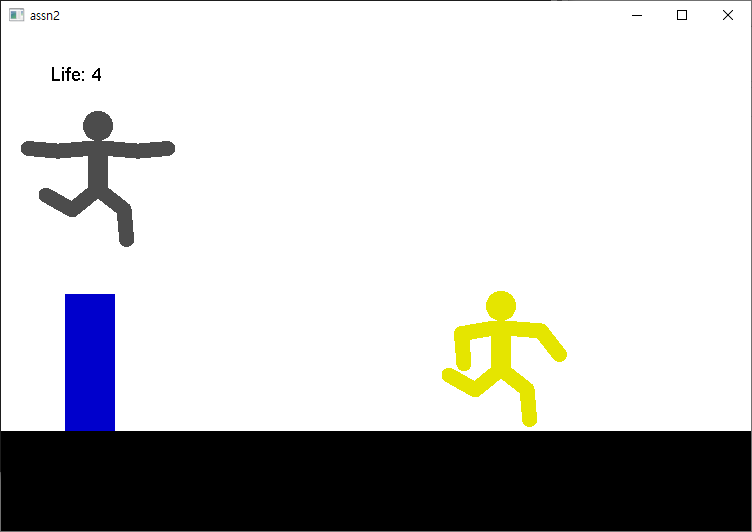
벽을 계속 성공적으로 통과하여 도둑과 가까워지다가, 플레이어와 도둑이 일정 이상 가까워지게 되면 콘솔에 ‘Win’이 출력되어 플레이어가 승리했음을 알리고, 추격이 종료된다.



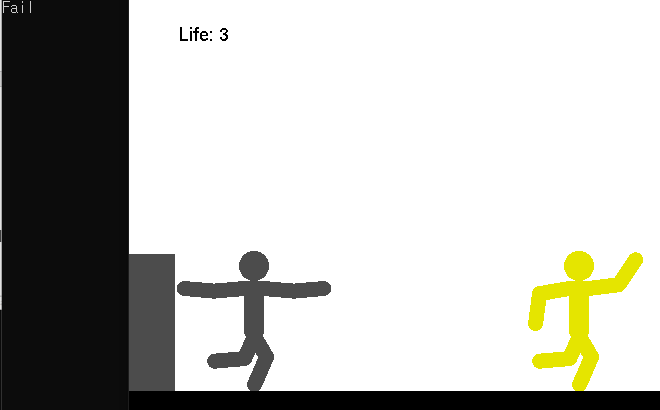
C를 누르면 콘솔에 ‘All pass’가 출력되어 All pass 치트가 활성화되었음을 알리고, 플레이어의 색과 무관하게 벽을 성공적으로 통과한 것으로 간주한다.



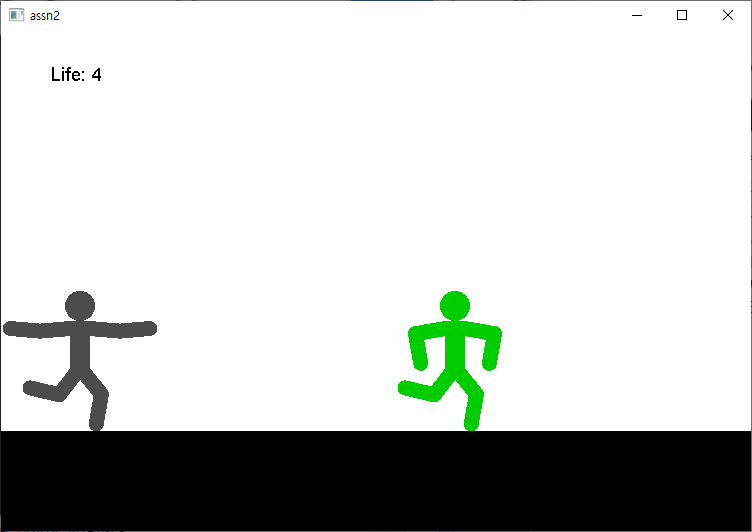
F를 누르면 콘솔에 ‘All fail’이 출력되어 All fail 치트의 활성화를 알리고, 플레이어의 색과 무관하게 벽을 통과하지 못한 것으로 간주한다.



스페이스바 키 입력을 통해 플레이어는 도둑과 같은 자세(색)을 취해 벽을 통과하는 대신 벽을 점프하여 넘어갈 수 있다. 대신, 플레이어는 도둑과 멀어지게 되며, 이 경우 플레이어와 도둑이 서로 너무 멀어져 한 화면에 담을 수 없게 되는 것을 방지하기 위해 카메라를 줌 아웃시켜 좀 더 넓은 영역을 비추게 한다. 콘솔에 Pass나 Fail이 출력되지는 않는다.



도둑이 일정 확률로 점프하여 벽을 지나갈 경우, 벽은 회색으로 그대로 남게 된다. 이 경우, 플레이어도 따라서 점프해야 하는데, 점프하지 않을 경우 통과하지 못한 것으로 처리되어 life가 1 감소하고, 콘솔에도 Fail이 출력된다.



플레이어도 점프하여 벽을 넘을 경우, 점프 액션의 영향으로 인해 플레이어와 도둑 모두 뒤로 조금 물러난다. 이 상황이 반복될 경우, 플레이어나 도둑이 화면 밖으로 벗어날 수가 있으므로, 플레이어와 도둑 둘 모두 점프한 경우 카메라를 왼쪽으로 이동시켜주었다. 콘솔에 Pass나 Fail이 출력되지는 않는다.

**Discussion**

1. 인간형 캐릭터 구현
2. openGL의 glMatrix와 glm

이번 과제에서는 openGL의 glMatrix 관련 함수들과 openGL의 Matrix stack을 활용하여 hierarchy를 가진 캐릭터를 구현하는 것이 주 목적이었다. 그에 따라 hierarchy의 구현에 LCRS 트리를 사용하였고, 이 때 LCRS 트리의 각 노드에는 현재의 matrix 오른쪽에 곱해질 행렬인 mtx와 그 이후에 또 추가적으로 행해질 관절의 회전변환을 나타내는 additionalTransform이 glm::mat4 타입으로 저장되어 있었다.

openGL의 matrix 관련 함수와 treeNode에 저장된 glm::mat4가 서로 호환이 되지 않았고, 이를 해결할 수 있는 방법에 대해 고민해 보았었다.

1. treeNode에 glm::mat4를 저장하는 대신 openGL의 matrix stack만을 활용한다.
2. openGL의 matrix 연산을 사용하지 않고 glm 연산만을 이용한다.
3. openGL의 matrix와 glm matrix를 호환시킬 방법을 찾는다.

첫 번째 방법은 구현하기 지나치게 어렵고 복잡하다고 판단하였고, 두 번째 방법 역시 수업에서 배운 openGL 행렬 연산을 사용하지 않는 방법이기에 사용하기 어려울 것이라고 판단하였다.

그리하여, 마지막 방법인 openGL과 glm을 호환시킬 방법을 탐색했고, 그 결과 glMultMatrixF(glm::value\_ptr(mtx)) 함수를 통해 treeNode에 필요한 변환 행렬을 glm::mat4 타입으로 저장하면서 openGL의 matrix 연산을 사용할 수 있게 되었다.

1. glMatrixMode

LCRS tree의 tree traversal을 통한 캐릭터의 구현을 처음 완료하였을 때, 다른 기능은 모두 정상적으로 작동했으나, 이전 assn1에서 구현해 놓았던 카메라 줌 기능이 이상하게 작동하였다. 처음에는 캐릭터 구현 과정에서 push/pop을 제대로 활용하지 못한 것을 원인으로 보았으나, 검색을 통해 Matrix Mode가 문제임을 알게 되었다.

openGL에는 GL\_MODELVIEW, GL\_PROJECTION, GL\_TEXTURE의 세 가지 matrix mode가 있고, GL\_MODELVIEW는 모델의 실제 위치를 정할 때, GL\_PROJECTION은 실제 화면에 어떻게 나타날지에 대해 정의하는 matrix mode임을 알게 되었다.

그에 따라, 캐릭터를 그려줄 때, 즉 캐릭터 모델링 시에는 matrix mode를 GL\_MODELVIEW로 설정하여 matrix 연산 및 push/pop을 시행하였고, 그 외의 경우에는 GL\_PROJECTION으로 설정하고 Projection 모드의 행렬은 단위행렬로 설정하여 카메라 줌 기능 이상 문제를 해결할 수 있었다.

1. 하체 애니메이션 구현

하체 애니메이션은 상체 애니메이션과 달리 하나의 애니메이션 루프가 게임 시작부터 종료까지 반복되는 형태이다. 이러한 애니메이션 루프를 하체 자세 A와 하체 자세 B를 정의해두고, 일정 시간마다 A에서 B로, 다시 B에서 A로 왕복하게끔 하는 방식으로 구현하였다.

부드러운 애니메이션 구현을 위해서는 매 프레임마다 하체 관절 각도를 재정의하여 캐릭터를 그려주어야 했는데, 이를 구현하기 위해 자세 A, B의 구현에 필요한 관절 각도를 미리 계산해두고, 매 프레임마다 현재 프레임에 취해야 할 자세에 맞게 선형 보간법으로 관절 각도를 계산하여 하체의 4가지 관절을 움직여주었다

이 때, 관절 각도를 편하게 계산하기 위하여 treeNode의 mtx를 매 프레임마다 재정의하는 대신, mtx를 캐릭터의 기본 자세를 정의하기 위한 행렬로써 사용하여 고정시키고, 기본 자세에서 추가적인 관절 회전만을 정의하기 위한 새로운 행렬인 additionalTransform을 정의하는 방식으로 관절의 움직임을 구현하였다.

1. 상체 애니메이션 구현

상체 애니메이션에서 가장 많은 시간을 소요한 부분은 어떠한 방식으로 애니메이션 변화를 구현할지 설계하는 단계였다. 총 initialPose와 색을 가진 4가지 포즈까지 총 5개의 포즈가 존재했으며 모든 포즈의 변화를 개별적으로 구현한다면 총 20가지 포즈를 구현해야 했다. 이를 간소화하기 위해 각 포즈 간의 차이를 계산하고 해당 차이만큼 포즈 변화 프레임동안 변화하게 하였다.

1. Jump 구현

Jump 액션을 구현하는데 가장 주요하게 고려해야할 점은 현재 구현된 프로그램과 어색하지 않게 적절한 값을 설정해주는 데 있었다. 기존의 프로그램에서 점프를 구현하기에 벽의 높이가 지나치게 높았으며, 벽의 이동속도가 충분하지 않을 경우 적절한 위치에서 점프를 한 뒤에도 벽을 넘지 못하는 상황이 연출되었다. 플레이어가 점프 입력을 너무 늦게 입력하거나 빨리 입력하여 시각적으로 점프에 실패한 상황에서도 벽을 통과하는 상황을 방지하기 위하여 점프 상황에서 플레이어가 벽을 통과할 때 기준으로 하는 높이를 설정해야 했다. 상하 좌우 운동에는 복잡한 구현이 사용되지 않았으나, 사실성을 높이기 위해 여러 변수들을 적절히 수정해야했다.

**Conclusion**

1. 인간형 캐릭터 구현

이번 assn2에서의 인간형 캐릭터 구현 과정에서 겪은 treeNode에서의 openGL과 glm matrix 호환 문제 해결 과정에서 openGL의 행렬 연산, glm 및 openGL의 matrix stack의 사용 방법을 익힐 수 있었다. 또한, 카메라 줌 오작동 문제의 해결 과정에서 openGL의 matrix에는 세 가지 모드가 존재하며, 각각의 모드를 적절한 상황에 맞게 사용하여야 함을 알게 되었다.

openGL의 사용에 좀 더 익숙해진 것과 더불어, hierarchy를 가진 오브젝트를 그래프(이 과제에서는 LCRS tree)를 이용해서 캐릭터를 정의해보는 과정을 통해 그래픽스에서의 model frame 정의, 행렬 연산, 그리고 hierarchy 개념 및 실제 구현의 이해에 도움이 되었다.

1. 애니메이션 구현

선형 보간으로 계층을 가진 개체에 대한 애니메이션을 구현하면서, 캐릭터의 애니메이션이 동작하는 과정을 이해할 수 있었다. 계층화에 오류가 발생하였을 때 캐릭터의 애니메이션에 결함이 발견되며, 초기 설계 단계에서 체계적이 계층화가 위와 같은 결함을 방지할 수 있다는 것을 배웠다.

1. Jump 구현

애니메이션에 사용되는 변수를 적절히 사용하는 것이 애니메이션의 세부 동작 구현만큼 중요하다는 것을 알 수 있었다. 실제 환경과 차이가 큰 변수를 사용하였을 때 사용자는 애니메이션이 부자연스럽다고 느끼게 되고, 이는 제작자의 의도와 다르게 보여지는 결과를 이끌어내게 된다. 초기 설계 단계에서 알고리즘 구현뿐만 아닌 구체적인 변수의 값 또한 구체적으로 설정해야함을 알았다.

**Direction of Improvement**

1. 포즈 변화에 따른 색 변화

현재 구현된 프로그램에서는 포즈가 변화할 때 물리적인 포즈가 모두 변화하기 전에 색의 변화를 통해 도둑이 어떤 포즈로 변화할 것인지 예측할 수 있다. 캐릭터의 색 변화 또한 애니메이션으로 점차적으로 변화하도록 구현한다면 물리적인 팔의 각도 변화량과 색의 변화량이 같게 유지되어 애니메이션의 질을 더 높일 수 있을 것으로 보여진다.

1. 애니메이션과 실제 물리 현상의 부조화

하체와 상체의 애니메이션의 부조화 또한 현실감을 저하시키는 요소 중 하나이다. 실제 캐릭터는 하체의 움직임에 따라 상체도 조화롭게 움직인다. 현재의 프로그램에서는 캐릭터가 계속 추격을 진행해 하체는 계속해서 움직이지만 상체는 고정되어 있다. 달리는 속도나 다리의 위치에 따라 팔과 몸통의 각도를 조정한다면 애니메이션을 조금 더 자연스럽게 구현할 수 있을 것으로 고려된다.

점프 액션을 수행하였을 때 캐릭터는 수직방향의 상하 운동을 수행하며 일정 거리만큼 이동 방향 반대로 움직이게 된다. 실제 환경에서 뛰어오르는 동작을 수행하면 중력가속도로 인해 수직방향으로 등가속도 운동을 하게 되지만, 현재 프로그램에서는 등속도로 상승하다가 특정 높이에 도달하면 등속도로 하강한다. 특정 높이에 도달하였을 때 속도가 변화하면서 충돌한 듯한 애니메이션이 연출되게 되며 이를 등속도 운동이 아닌 등가속도 운동으로 구현한다면 점프 액션을 수행하였을 때의 부자연스러움을 줄일 수 있을 것이다.