CSED451 Assignment #2

2D ANIMATION 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. Coordinates (coordinates.h)

플레이어가 벽을 통과함에 따라, 카메라를 줌인하기 위해 카메라의 영역을 정의하는 클래스이다. 멤버 변수로는 카메라 상하좌우의 좌표 값이 있다.

1. 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, y를 가지고 있다. 이들은 각각 오브젝트 타입(0일 경우 rect, 1일 경우 character), 오브젝트의 색깔, 오브젝트를 그릴 때 필요한 x, y좌표에 해당한다.

1. Rect: public object (rect.h)

Object를 상속받아 맵 상의 직사각형 오브젝트(바닥, 벽)을 정의하는 클래스이다.  
rect 오브젝트들은 공통적으로 type이 0이며, x, y는 직사각형의 왼쪽 아래 점의 좌표를 의미한다. 추가적으로 width, height라는 멤버 변수를 가지는데, 이는 각각 직사각형의 가로 길이, 세로 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, y는 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
2. 화면 상 오브젝트의 출력

화면 상 모든 출력은 display() 함수에서 이루어진다.

rect::draw(), character::draw(), writeLife() 등 화면에 표시되는 오브젝트의 종류마다, 그리고 화면에 적히는 글씨마다 출력에 필요한 행동이 다르기에 각각을 별개로 함수로 선언하고, 이러한 함수들을 display()함수에서 출력이 필요한 오브젝트마다, 그리고 출력이 필요한 글씨마다 호출하는 방식으로 구현하였다.

1. 벽의 생성 및 이동

벽의 생성과 이동은 매 frame(1/60초)마다 호출되는 moveWall() 함수에서 이루어진다.

moveWall 함수에서는 벽의 이동 속도인 wallSpeed에 맞게 벽을 이동시켜주고, 벽이 플레이어 혹은 도둑 캐릭터와 충돌했는지를 판정하여, 벽이 도둑과 충돌한 경우 벽의 색을 현재 도둑의 색으로 변경시켜주고, 벽이 플레이어와 충돌한 경우 플레이어의 색과 벽의 색을 비교하여 pass/fail 여부를 확인하고, 그에 맞는 action을 수행한다.

벽이 화면 밖(왼쪽)으로 벗어난 경우, 벽을 초기 위치로 이동시켜주어 벽의 재생성을 구현하였다.

마지막으로, 게임의 현재 상황에 맞게 game status(IDLE, LOSE, WIN, PASS, JUMP, FAIL)를 리턴한다.

1. 충돌 판정

오브젝트들의 충돌 판정은 object::collisionCheck(object\* a) 함수에서 이루어진다. 오브젝트의 충돌은 크게 벽-플레이어/도둑의 충돌, 그리고 플레이어-도둑의 충돌 2가지 경우로 나뉜다.

벽과 플레이어 혹은 벽과 도둑의 충돌은 벽의 왼쪽 모서리의 x좌표가 캐릭터 torso의 중심의 x좌표를 지나는 순간 충돌했다고 판정하고, 캐릭터와 벽이 겹치는 동안 충돌의 중복 판정을 방지하기 위해 캐릭터의 isCollided를 true로 만들어주어서 중복 판정을 방지하였다.

플레이어와 도둑의 충돌은 두 캐릭터의 x 거리가 15보다 작을 때 충돌하였다고 판정하였다. 플레이어와 도둑이 충돌 했을 때 콘솔 창에 Win을 출력하고 게임을 종료한다.

1. 플레이어 캐릭터의 조작

플레이어의 조작 구현은 방향키 입력을 받는 glutSpecialFunc를 이용하여 구현하였다. 위키를 누르면 빨간색, 아래 키를 누르면 초록색, 왼쪽 키를 누르면 파란색, 오른쪽 키를 누르면 플레이어 캐릭터가 노란색이 되도록 구현하였다. 그리고, 후술할 점프 기능은 스페이스바 키를 누르면 플레이어가 점프하여 벽을 뛰어넘도록 구현되었다.

1. 치트 기능

치트 기능은 glutKeyboardFunc를 이용하여 구현하였다. 과제의 지시사항대로 ‘c’를 입력하면 all pass 모드로, ‘f’를 입력하면 all fail 모드로 진입시키고 콘솔에 출력시켜 치트를 사용했음을 알 수 있게 구현하였다.

1. Pass/Fail 시의 Action
2. Pass Action

collisionCheck에서 wall과 player가 충돌할 때, allPass의 값이 1이거나 wall의 색과 player의 색이 같을 때 pass로 간주한다. 플레이어가 Jump 액션을 통해 벽을 통과한 것은 pass로 간주하지 않는다. 콘솔 창에 pass를 출력하고, wallSpeed를 wallSpeed Increment만큼 증가시킨다. 75로 초기화한 도둑의 포즈 변화 주기 posePeriod를 5만큼 감소시킨다. 50프레임 동안 플레이어의 위치가 movingDistance만큼 오른쪽으로 이동하게 하기 위하여 playerNewX에 moving Distance를 더해준다. 이후 매 프레임마다 character :: checkNewPosition() 함수를 호출하여 player의 x좌표가 playerNewX 보다 작은 지 확인하며, 작을 경우 player의 x 좌표를 player.moveright 함수로 distancePerFrame만큼 이동시킨다. 카메라 또한 zoomFrame 동안 coordinatesIncrement만큼 줌인하기 위하여 newWorld을 world 값에 coordinatesIncrement 값을 더한 값으로 한다. 매 프레임마다 world의 right 값이 newWorld의 right 값보다 큰지 확인하며, 클 경우 world의 상하좌우 좌표에 incrementPerFrame 좌표를 더해준다. 이후, gluOrtho2D에 새로운 world 값을 넣어 바뀐 world 좌표로 줌인되게 된다.

카메라가 줌인되더라도 좌측 상단의 lifeText의 위치는 고정되어 있어야 한다. lifeText의 좌표는 다음 식을 따르며, frameMoved는 처음 화면으로부터 몇 프레임만큼 줌인되었는지 확인한다.

lifeX = world.getLeft() + LIFE\_DEFAULT\_X \*

(world.getRight() - world.getLeft()) / WORLD\_SIZE\_X;

lifeY = world.getBottom() + LIFE\_DEFAULT\_Y \*

(world.getTop() - world.getBottom()) / WORLD\_SIZE\_Y;

LIFE\_DEFAULT\_X, LIFE\_DEFAULT\_Y는 처음 화면의 life의 좌표이며, WORLD\_SIZE\_X, WORLD\_SIZE\_Y는 처음 화면의 world의 너비와 높이이다.

1. Fail Action

collisionCheck에서 wall과 player가 충돌할 때, allFail의 값이 1이거나 wall의 색과 player의 색이 다를 때 fail로 간주한다. 도둑이 벽을 jump 액션으로 통과하였을 때, 플레이어가 jump 액션으로 통과하지 않거나, jump가 기준 높이보다 낮은 경우 fail로 간주한다. 콘솔 창에 Fail을 출력하고 life를 1만큼 감소시킨다. life가 0이 되면 콘솔 창에 Lose를 출력하고 게임을 종료한다.

1. 인간형 캐릭터 구현
2. LCRS tree(treeNode.h)

이번 과제에서 인간형 캐릭터는 최소 8개 이상의 관절과 그에 해당하는 hierarchy를 가지는데, 이 때 인간형 캐릭터의 hierarchy를 표현하기 위해 필요한 Left Child Right Sibling(LCRS) tree를 구현하는 클래스이다.

각 노드는 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.

이 LCRS tree를 순환하여 실제로 화면상에 캐릭터를 그려주는 함수인 traverse(treeNode \*current) 함수는 character.cpp에 구현되어 있다. glPushMatrix와 glPopMatrix를 이용하고, sibling 및 child 노드에서 traverse 함수를 자체적으로 재귀호출하여 hierarchy를 구현하였다.

1. 캐릭터 model 구현(character.cpp)

이번 과제에서 캐릭터는 head, torso, lua, rua, lul, rul, lla, rla, lll, rll 총 9개 부분으로 나누어져 있고, 이들을 scene graph hierarchy를 이용하여 구현해야 한다. 이를 위해서 9개의 파츠 각각을 그리기 위한 3가지의 모델 프레임을 정의하였다.

1. Head

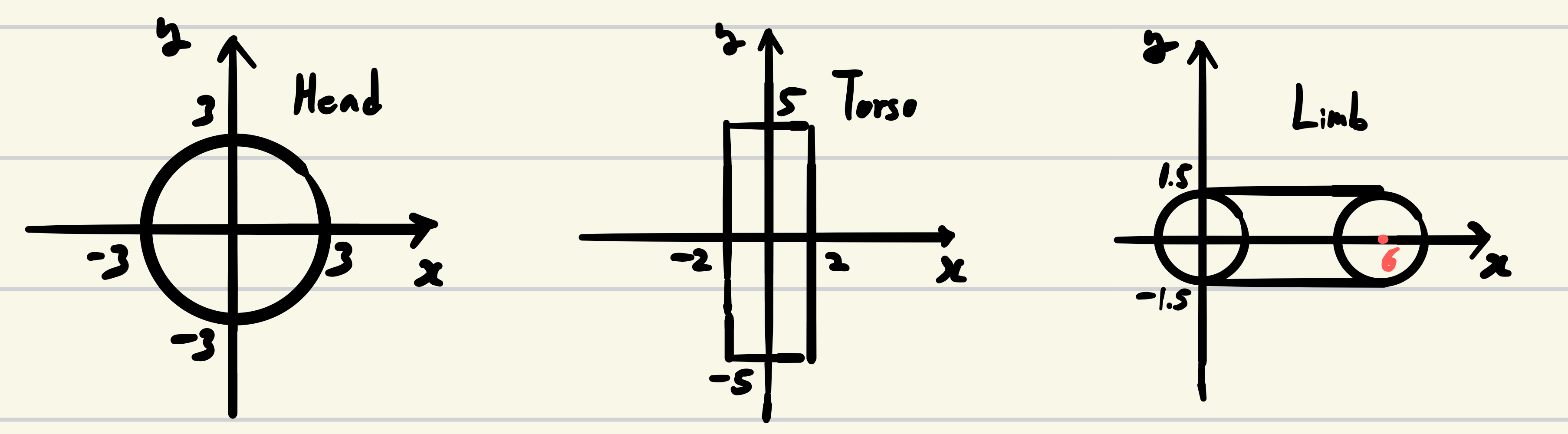
머리 모델은 반지름 3의 (0,0) 중심인 원 모양으로 나타난다.

1. Torso

몸통 모델은 중심이 (0,0)이고 폭 4, 높이 10의 직사각형 모양으로 나타난다.

1. Limb

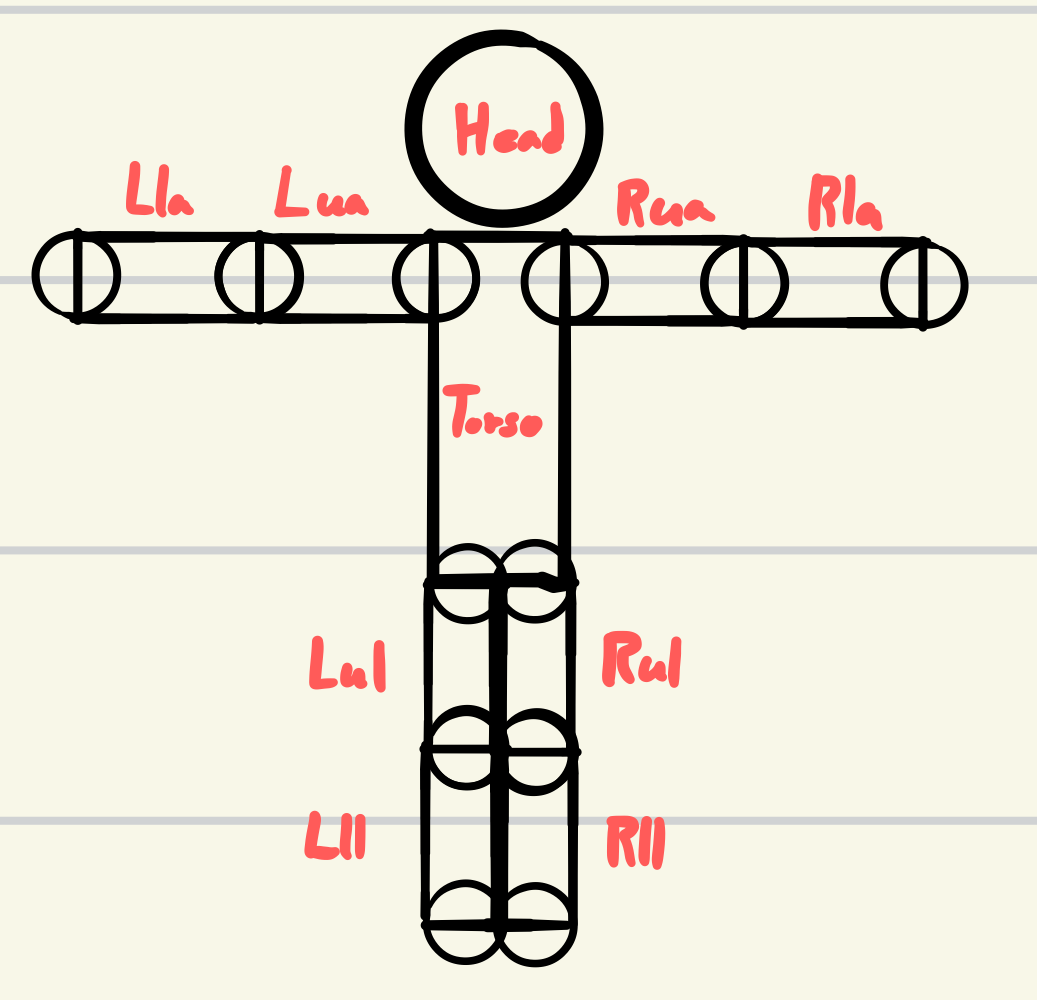
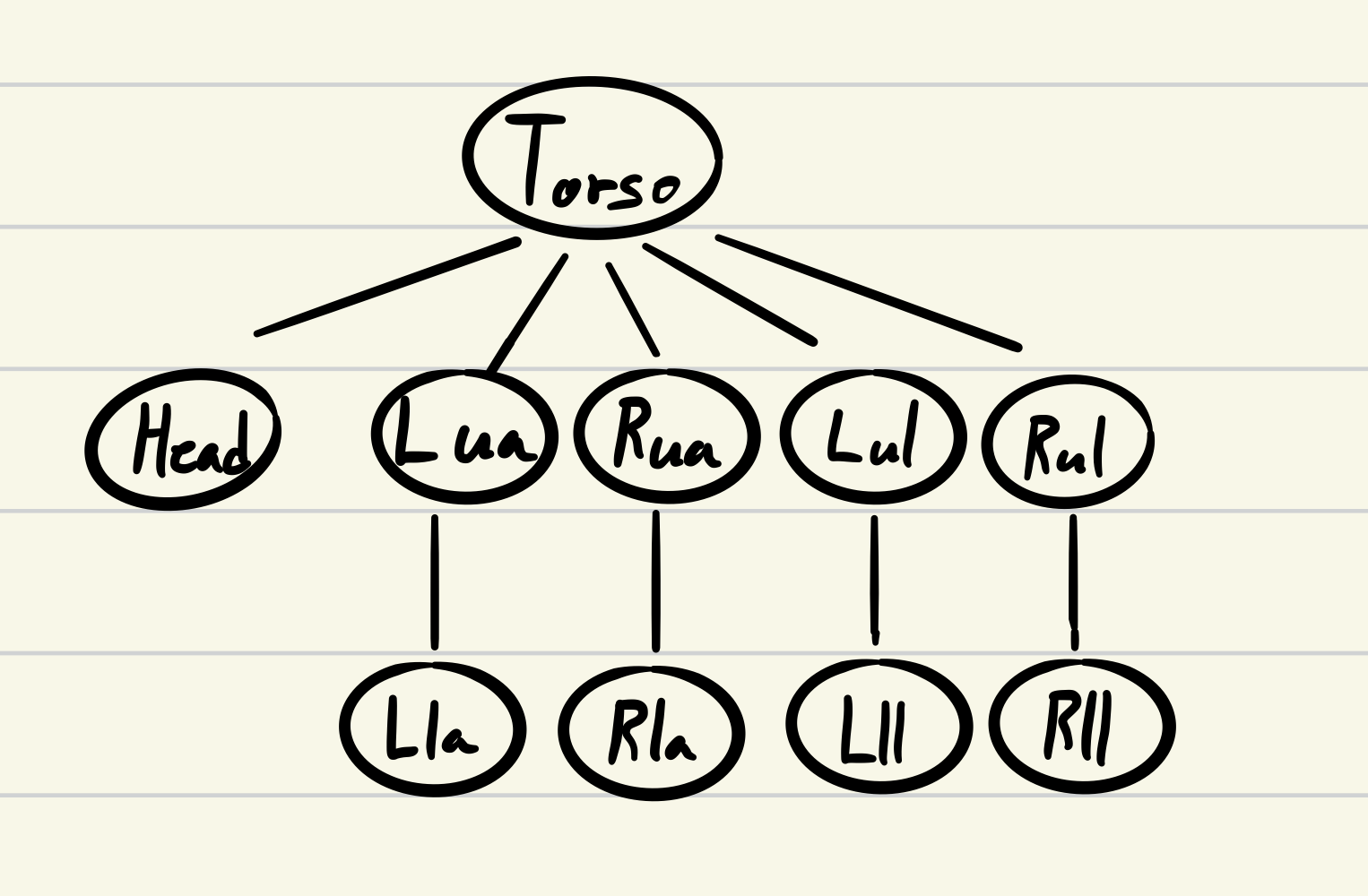
사지 모델은 폭 6, 높이 3의 직사각형 양 끝에 반지름이 1.5인 원이 달린 모양으로 나타난다. Lul, rua, lul, rul, lla, rla, lla, rll 가 각각 하나의 limb에 해당한다.



이러한 모델 프레임의 실제 구현은 character.cpp의 drawHead, drawLimb, drawTorso 함수에 구현되어 있다. 이 때 LCRS tree를 구현한 treeNode의 함수 포인터인 draw가 각각 캐릭터 파츠 종류에 따라 이 3가지의 draw 함수를 나타내게 된다.

1. Hierarchy를 가진 인간형 캐릭터 구현(character.cpp)

캐릭터는 head, torso, lua, rua, lul, rul, lla, rla, lll, rll 노드를 가지고 있으며, 이들의 hierarchy는 아래 그림과 같다. Hierarchy는 캐릭터를 생성할 때 constructor에서 정의되며, 캐릭터 클래스의 constructor에서 정의된 기본적인 캐릭터 모양은 아래와 같다.



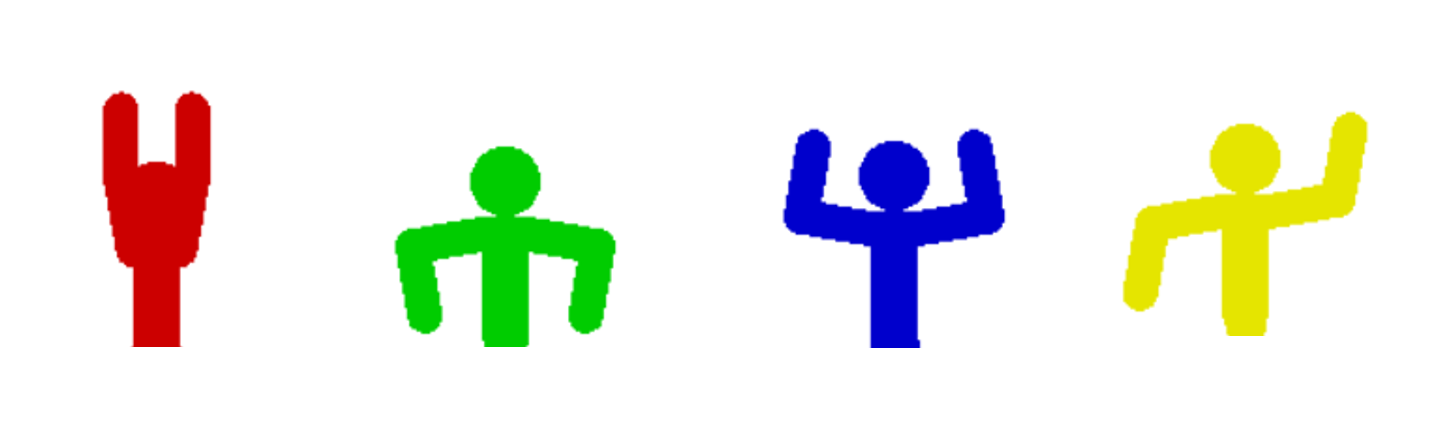
위 그림에서 rua는 limb 모델을 적절한 위치로 옮긴 것이고, lua는 limb 모델에 180도 회전변환을 적용하여 적절한 위치로 옮긴 것이고, lul, rul은 limb 모델에 -90도 회전변환을 적용하여 적절한 위치로 옮긴 것이다.

이러한 기본적인 모양은 캐릭터의 각 파츠별 node의 mtx에 constructor에 의해 저장되어 있으며, 이 기본 상태에서의 추가적인 관절 회전은 각 node의 additionalTransform에 저장된다. additionalTransform은 매 프레임마다 변경될 수 있으며, 애니메이션 구현은 매 프레임마다 캐릭터를 화면에 그리는 character::draw() 함수 내부에서 각 node의 additionalTransform을 현재 angle에 맞게 재정의하고, LCRS tree traversal 과정에서 재정의된 additionalTransform을 기본 mtx에 추가적으로 반영하는 방식으로 구현되어 있다.

1. Pose 구현(pose.h)

플레이어와 도둑은 총 4가지 포즈를 취할 수 있다. 포즈는 상체의 형태만으로 결정되며 각 포즈에 따른 색이 mapping되어 있다. 포즈의 형태는 팔의 각도가 결정한다. 각각의 포즈에 따라 lua, lla, rua, rla의 각도를 가지며 각 포즈에 따른 정보는 다음과 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | lua | lla | rua | rla | color |
| initialPose | -5 | -10 | -5 | 10 | grey |
| poseA | -80 | -10 | 80 | 10 | red |
| poseB | 10 | 90 | -10 | -90 | yellow |
| poseC | -10 | -90 | 10 | 90 | blue |
| poseD | 10 | 70 | 10 | 70 | green |



1. 하체 애니메이션 구현

하체 애니메이션은 character.cpp의 character::lowerBodyAnimation(int currentFrame, int period) 에 구현되어 있다. 이 함수는 전체 period 프레임으로 구성된 하체 애니메이션 루프에서 currentFrame에 해당하는 번째의 프레임을 그려주는 함수로, 현재는 main.h에 의해 총 60프레임으로 작동하고, 매 프레임마다 호출되게끔 되어있다.

하체 애니메이션은 상체와 달리 게임 전체에서 한 가지 루프가 지속되는 형태로, 하체가 달리는 것처럼 보이기 위한 시작 자세와 끝 자세를 미리 정의해두고, 그 자세를 만들기 위한 하체 node(lul, lll, rul, rll)의 additionalMatrix에 추가로 적용될 회전 각도를 character.h에 정의해두었다. 그리고, character::lowerBodyAnimation 함수에서는 전체 왕복 주기 중에서 현재 프레임에 맞는 자세의 angle을 계산하여 하체의 4가지 관절 각도를 변경시키는 방식으로 구현하였다.

1. 상체 애니메이션 구현

상체 애니메이션은 character.cpp의 character::upperBodyAnimation()에 구현되어 있다. 이 함수는 character::changePose(pose inputPose)가 실행되어 poseFrameCheck가 0으로 할당되었을 때 실행된다.

changePose는 도둑이 포즈 변화 주기에 따라 포즈를 변화하거나, 플레이어가 키보드 입력에 의해 포즈를 변화할 때 호출된다. 도둑의 포즈는 posePeriod마다 변화한다. 현재 해당 캐릭터가 갖고 있는 포즈와 변화시키려는 포즈가 다를 때 포즈가 변화한다. newPose에 변화할 pose인 inputPose를 업데이트하고, 애니메이션을 구현하기 위해 포즈 변화량인 poseVariance를 newPose에서 currentPose값을 뺀 값으로 할당한다. 현재 캐릭터의 색 또한 inputPose의 색으로 바꾸며, poseFrameCheck에 0을 할당하여 upperBodyAnimation 함수가 동작을 수행할 수 있도록 한다.

upperBodyAnimation 함수에서는 poseChangeFrame 동안 포즈를 변화시키는 동작을 수행한다. 현재 프로그램에서는 30 프레임동안 포즈가 변화한다. changePose 함수에서 계산한 poseVaraince를 poseChangeFrame 값으로 나눈 만큼 currentPose를 업데이트하며 poseFrameCheck값을 1씩 증가시킨다. poseFrameCheck 값이 30이 되면 더 이상 포즈를 변환하지 않으며, newPose와 currentPose가 같은 값이 된다.

1. Jump 구현

플레이어와 도둑은 점프 액션을 통해 벽을 넘을 수 있다. 플레이어는 사용자가 space키를 입력하면 jump를 실행하고, 도둑은 벽과의 거리가 일정 거리에 도달하였을 때 thiefJumpProbability 값에 따라 점프할지 판단한다. 도둑이 점프 액션을 통해 벽을 통과하면 도둑의 색과 무관하게 벽의 색은 변하지 않는다.

캐릭터가 jump를 수행하면 character::jump() 함수가 호출된다. jump 함수에서는 캐릭터가 점프 액션 후에 도달하게 되는 newX를 현재 x 좌표에서 jump\_back 만큼 뺀 값으로, newY를 현재 y좌표에서 jump\_height 만큼 더한 값으로 설정한다.

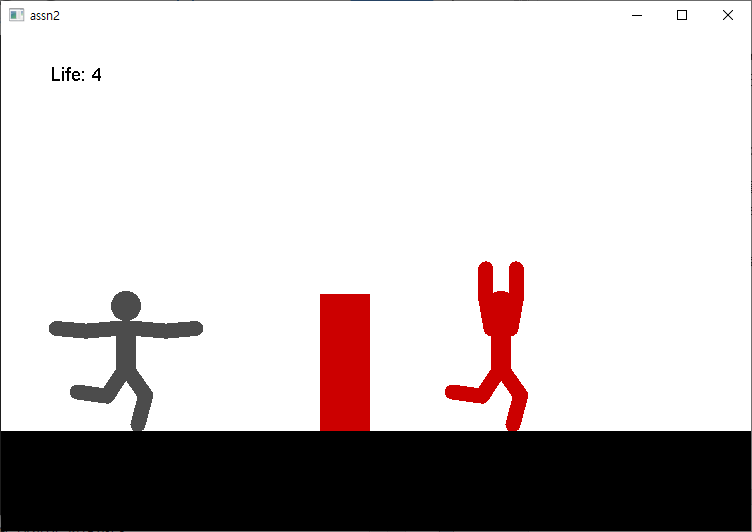
character::checkNewPosition() 함수에서는 newX 값이나 newY 값이 캐릭터의 좌표와 다를 때 해당하는 방향으로 캐릭터를 움직인다. 점프를 하는 동안 소요되는 프레임은 jumpFrame에 의해 결정되며 현재 100으로 설정되어 있다. 점프 높이는 40, 점프 액션으로 인해 캐릭터가 왼쪽으로 이동하는 거리는 10만큼 설정되어 있다. 50 프레임동안은 40만큼 y좌표가 증가하며, 남은 50프레임동안 40만큼 y좌표가 감소한다.

1. 카메라 줌 아웃, 카메라 이동 구현(추가 기능)

이번 과제에서 추가된 점프 액션은 플레이어와 도둑을 뒤로 이동시킨다. 이로 인해 점프가 거듭되면 플레이어 혹은 도둑이 화면 밖으로 나가거나, window가 너무 좁은 구역을 보여줘서 캐릭터와 도둑을 한번에 담을 수 없게 되는 일이 발생할 수 있어서 이를 예방하기 위해 카메라 줌 아웃 기능과 카메라 이동 기능을 추가로 구현해보았다.

1. 카메라 줌 아웃

도둑은 점프를 하지 않았지만 플레이어가 점프로 벽을 통과하는 경우, 플레이어와 도둑 사이의 거리는 멀어지게 된다. 이러한 현상이 반복될 경우, 현재 window가 지나치게 좁은 구역을 비춰 한 화면에 플레이어와 도둑을 모두 담지 못하게 될 수도 있다. 이를 예방하기 위해 도둑은 점프하지 않고 플레이어만 점프로 벽을 통과할 때 화면을 줌 아웃시켜 더 넓은 구역을 비추게 하는 기능을 추가하였다.

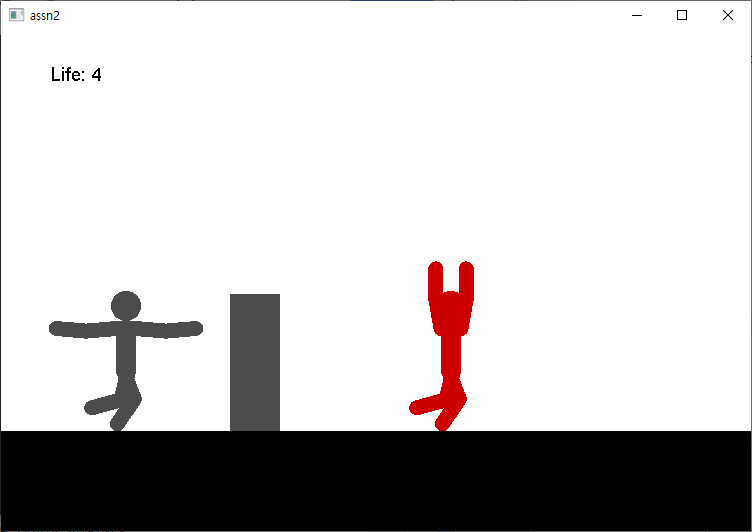
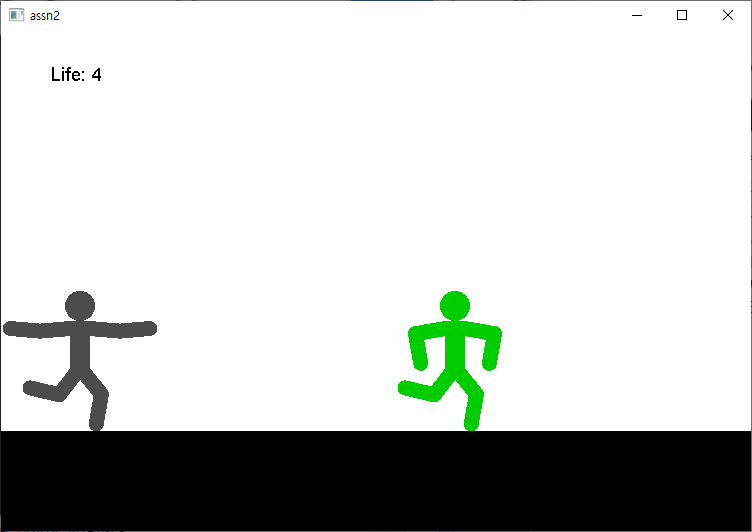
 🡪 

플레이어만 점프한 경우, 줌 아웃을 통해 더 넓은 영역을 비추는 모습이다.

구현 방식은 카메라 줌인과 동일한 방식으로 구현하였다. 카메라를 zoomFrame 동안 coordinatesDecrement만큼 줌 아웃하기 위하여 newWorld을 world 값에 coordinatesDecrement 값을 더한 값으로 한다. 매 프레임마다 world의 left 값이 newWorld의 left 값보다 큰지 확인하며, 클 경우 world의 상하좌우 좌표에 decrementPerFrame 좌표를 더해준다. 이후, gluOrtho2D에 새로운 world 값을 넣어 바뀐 world 좌표로 줌 아웃되게 된다.  
좌측 상단 lifeText의 위치 고정도 카메라의 줌인과 똑같은 방식으로 구현되어 있다.

1. 카메라 이동

도둑과 플레이어가 모두 점프하여 벽을 뛰어넘을 경우, 플레이어와 도둑 모두 뒤로 일정 거리를 이동하게 되고, 이것이 반복되면 플레이어와 도둑이 화면 밖으로 넘어가버릴 수 있다. 이를 예방하기 위해 도둑과 플레이어가 모두 점프할 경우, 카메라를 그만큼 뒤로 이동시켜주어 캐릭터가 화면 밖으로 나가는 현상을 예방하는 기능을 추가하였다.

 🡪 

도둑과 플레이어가 모두 점프할 경우 두 캐릭터 모두 뒤로 조금 밀려난 모습이다.

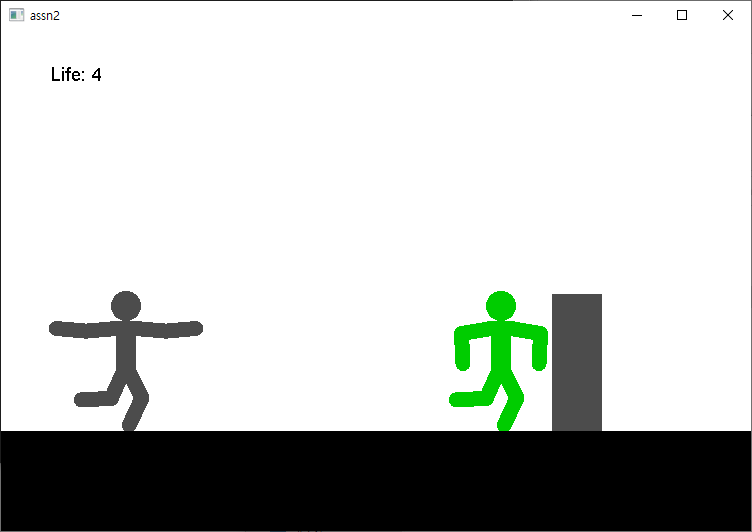
카메라 이동도 카메라 줌 인/아웃과 동일한 방식으로 구현하였다. 카메라를 zoomFrame동안 coordinatesMoveCameraLeft 만큼 왼쪽으로 이동시키기 위하여 newWorld을 world 값에 coordinatesMoveCameraLeft 만큼 더한 값으로 하였다. 매 프레임마다 world의 right와 left가 모두 newWorld의 right, left보다 큰지 확인하며, 클 경우 world의 상하좌우 좌표에 moveCameraLeftPerFrame 좌표를 더해주고 gluOrtho2D를 적용시켜 1프레임 단위의 카메라 이동을 구현하였다.  
좌측 상단 lifeText의 위치 고정도 카메라의 줌인과 똑같은 방식으로 구현되어 있다.

**How to Run**

프로젝트 폴더의 \bin\x64 폴더로 이동하여 graphics\_assn2.exe를 실행시킨다.

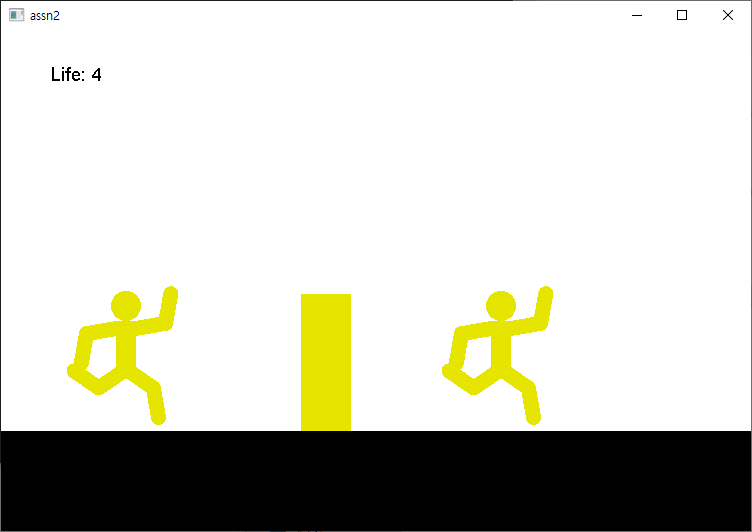
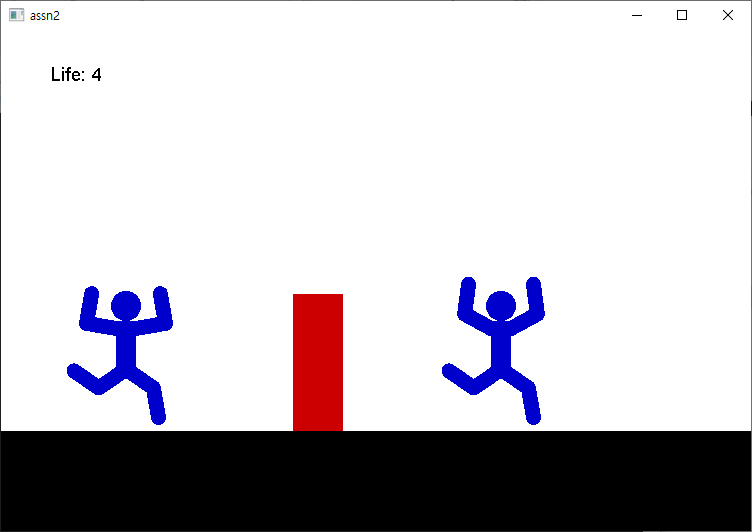
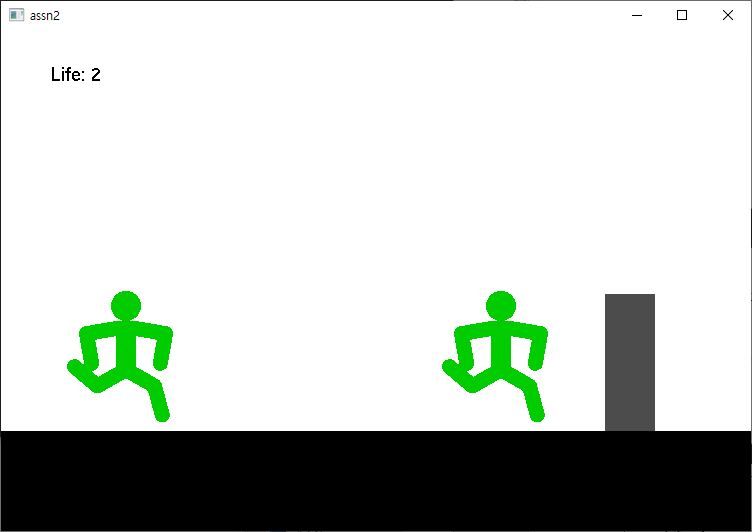
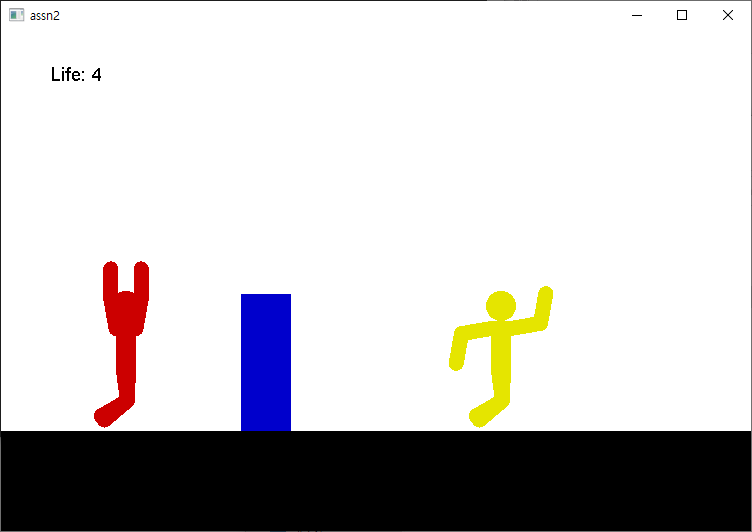
캐릭터의 포즈 변경은 방향키로 입력하며, 위, 아래, 왼쪽, 오른쪽 방향키는 각각 플레이어 캐릭터를 적색, 녹색, 청색, 황색으로 변경시킨다.

**Example**

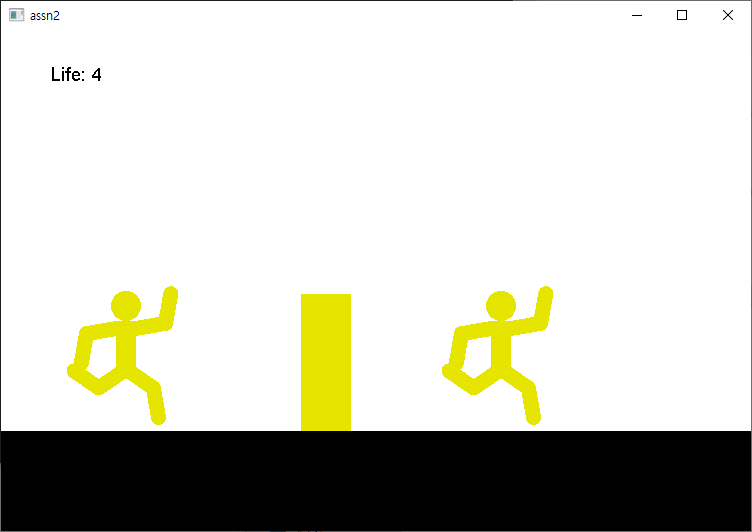


프로그램 실행 시 화면으로, 좌측 상단에 현재 life가 표시되고, 플레이어는 화면 왼쪽, 도둑은 화면 오른쪽에 위치한다. 벽은 화면 오른쪽에서부터 생성되어 왼쪽으로 이동한다.

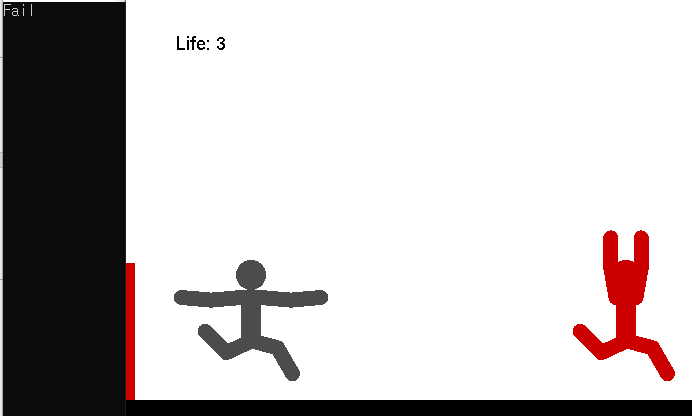
플레이어와 도둑 캐릭터 모두 부드럽게 달리는 모양의 하체 애니메이션 루프를 통해 사람이 달리는 듯한 모습을 그려낸다.



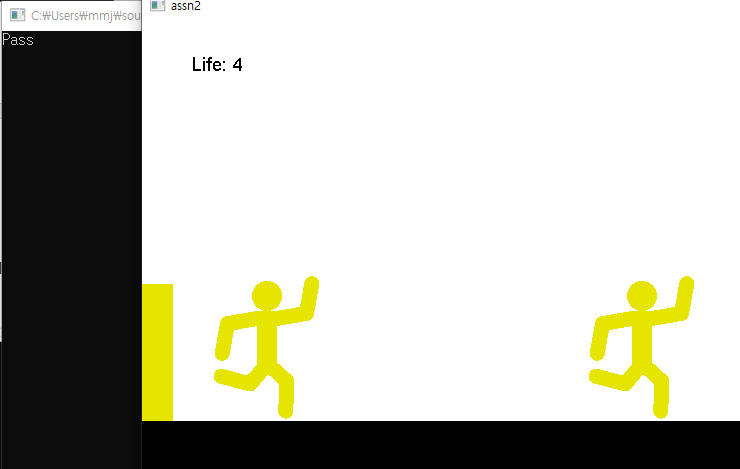
위(적), 아래(녹), 왼쪽(청), 오른쪽(황) 방향키를 눌러 플레이어의 색과 자세를 변경할 수 있다. 이 때, 상체 자세의 관절 전환은 부드럽게 처리된다.



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

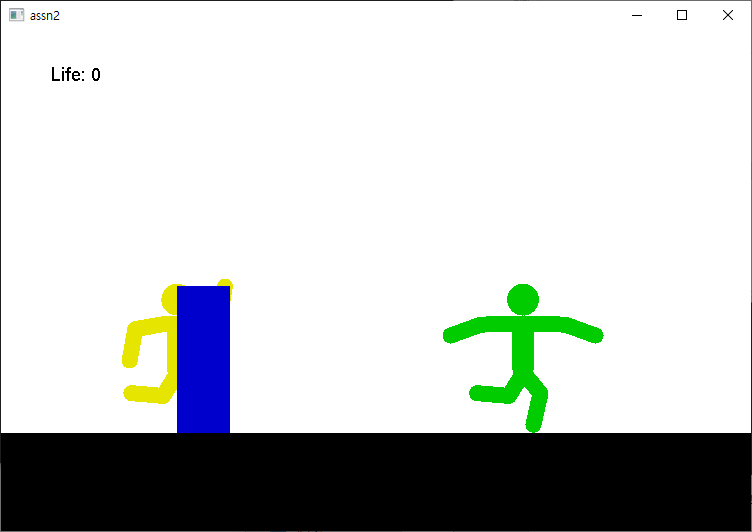
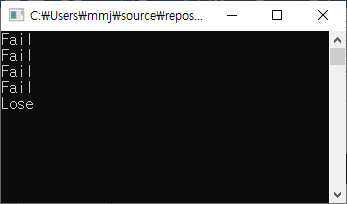


벽이 플레이어를 통과할 때 벽의 색과 플레이어의 색이 다를 경우, Life가 1 줄어들고, 콘솔에는 Fail이 출력된다.

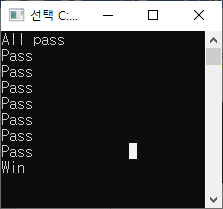


벽이 플레이어를 통과할 때 벽의 색과 플레이어의 색이 같을 경우, 콘솔에 ‘Pass’가 출력되고, 플레이어와 도둑의 거리가 가까워짐과 동시에 화면이 줌인되어 더 좁은 영역을 그린다.

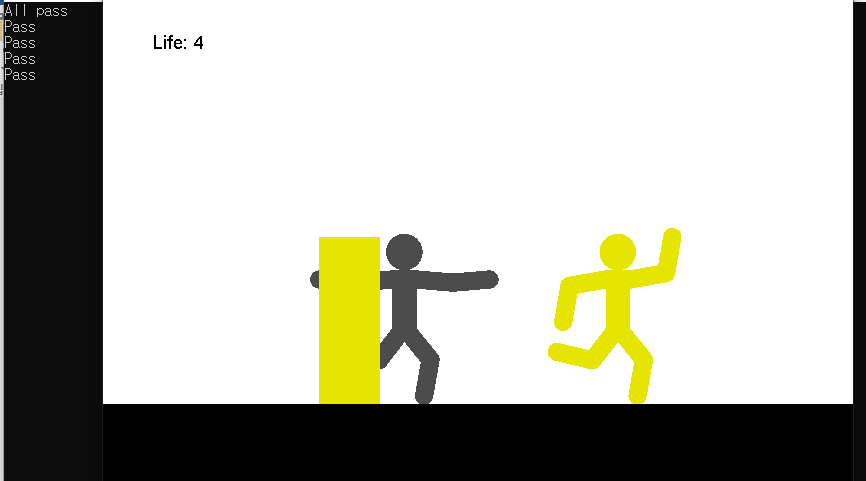
그리고 이후에 생성되는 벽의 속도와 재생성 간격이 빨라진다.



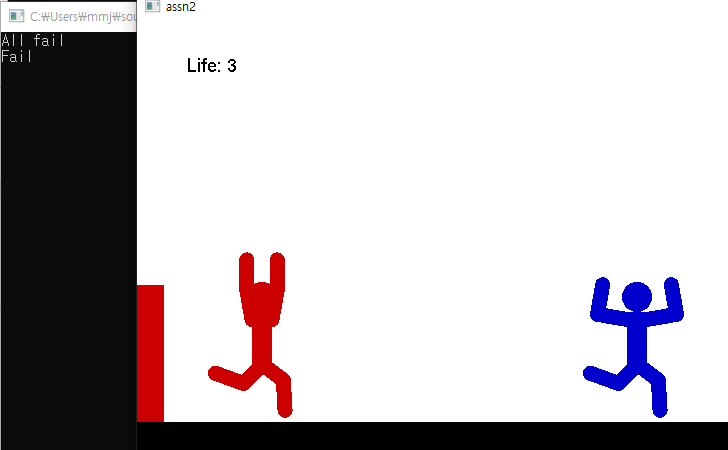
Life를 모두 잃어 0이 되면, 콘솔에는 Lose가 출력되고, 화면이 멈추어 추격이 종료된다.



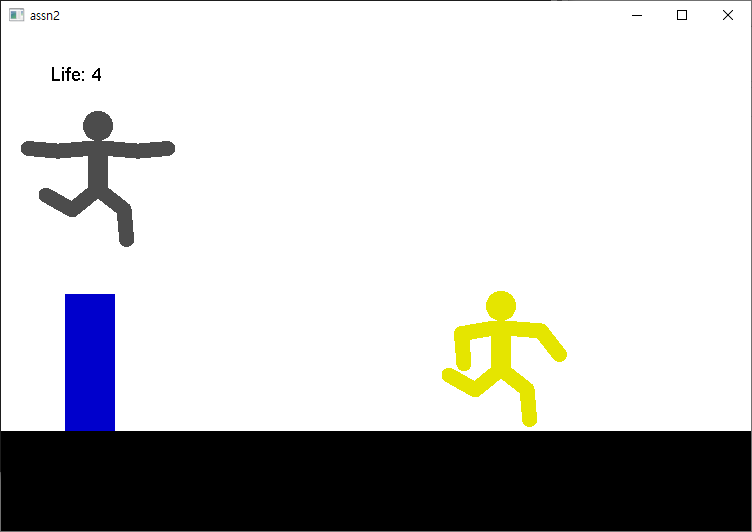
벽을 계속 성공적으로 통과하여 도둑과 가까워지다가, 플레이어와 도둑이 일정 이상 가까워지게 되면 콘솔에 ‘Win’이 출력되어 플레이어가 승리했음을 알리고, 추격이 종료된다.



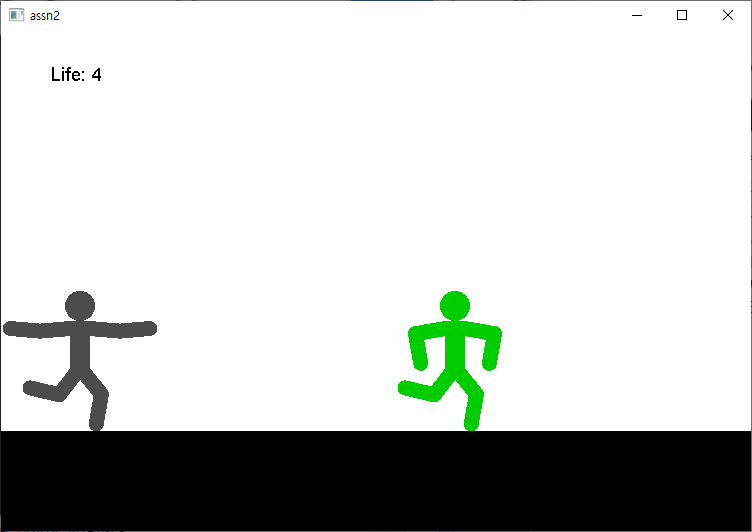
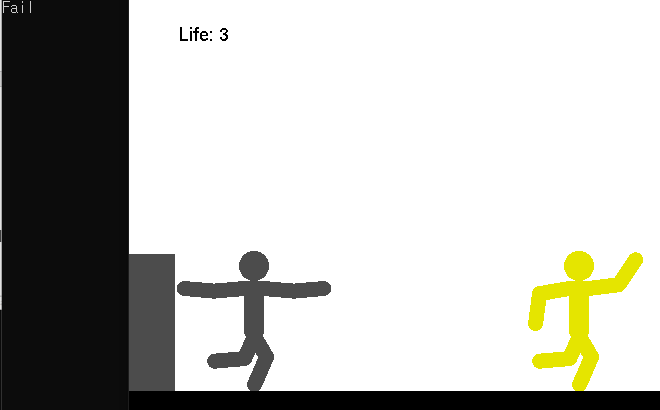
C를 누르면 콘솔에 ‘All pass’가 출력되어 All pass 치트가 활성화되었음을 알리고, 플레이어의 색과 무관하게 벽을 성공적으로 통과한 것으로 간주한다.



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 이 경우, 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.