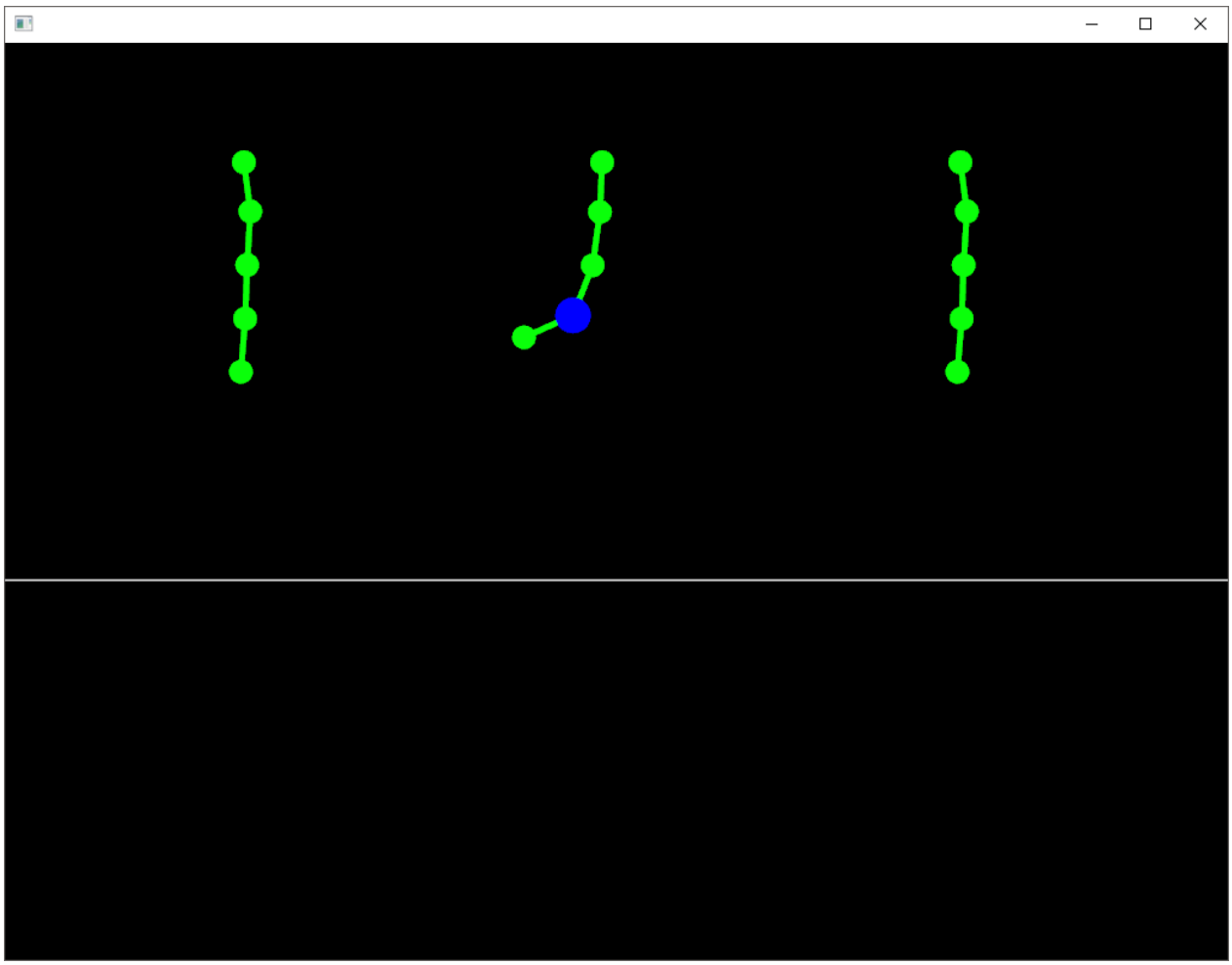
Prototyp mechaniki bujania się na linach z użyciem Spring-mass System.



GitHub: https://github.com/vanbear/SpringMassSystem
YouTube: https://youtu.be/d3vcOy9srEk

Początkowe założenia:

- stworzyć prototyp mechaniki do gry przy użyciu spring-mass system
- postać gracza z prowizoryczną fizyką
- zawieszone na statycznych punktach liny złożone z wielu punktów połączonych sprężynami
- możliwość złapania przez gracza liny i wprawiania jej w ruch

Co udało się zrobić?

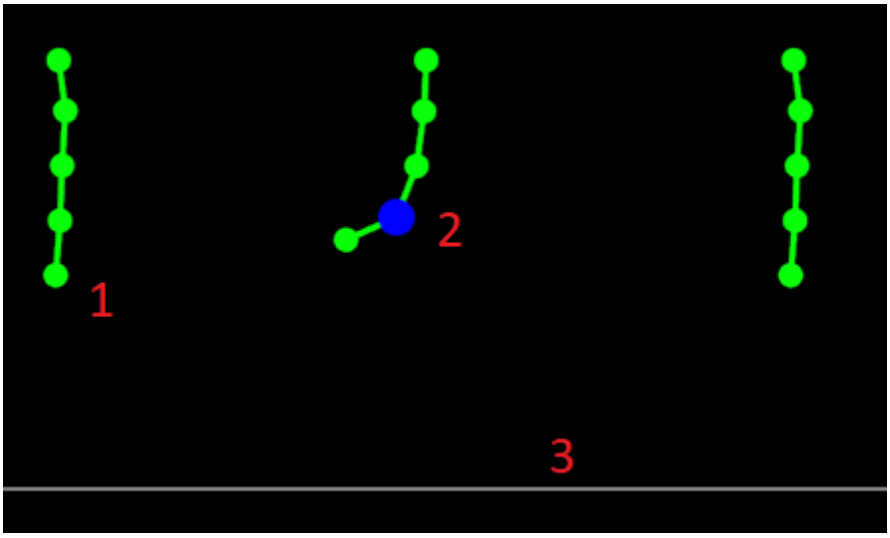
Wszystko!

Ważne informacje:

- Klawisz P uruchamia tryb debugowania
- Klawisze A i D służą do poruszania postacią na boki
- Klawisz W służy do skoku
- Klawisz E służy do łapania liny
- Zeskoczyć z liny możemy używając klawisza W

Opis projektu:

Projekt składa się z 3 elementów:



1) lina
2) gracz
3) podłoga

Lina:

Składa się ona z jednego statycznego punktu i 4 punktów swobodnych. Każdy punkt połączony jest ze swoimi sąsiadami sprężyną. Zarówno punkty jak i sprężyny są obiektami.

Obiekt punktu posiada wektory: poprzednie położenie, obecne i następne, prędkości i siły. Posiada również masę i zmienną logiczną isStatic, która definiuje nam, czy chcemy aby ten punkt się poruszał czy nie.
Przy deklaracji punktu poprzednią i obecną pozycję ustawiam na x i y podane w konstruktorze:

```
v_positionOld = { x,y };
v_position = { x,y };
```

Pierwsze dwa kroki aktualizacji położenia wykonuję metodą Eulera:

```
void Point::updateEuler()
{
    if (!m_isStatic)
    {
        v_velocity += v_forces * dt;
        v_positionNew = v_position + v_velocity * dt;
        v_positionOld = v_position;
        v_position = v_positionNew;
    }
}
```

Wszystkie następne kroki obliczam metodą Verleta:

```
void Point::updateVerlet()
{
    if (!m_isStatic)
    {
        v_positionNew = 2 * v_position - v_positionOld + (dt*dt*v_forces / m_mass);
        v_positionOld = v_position;
        v_position = v_positionNew;
    }
}
```

```
// aktualizacja położeń
if (counter < 2)
    for (auto &p : myPoints)
    {
        p->updateEuler();
        counter++;
    }
else
    for (auto &p : myPoints)
    {
        p->updateVerlet();
    }
```

<< Gdzie:

counter liczy ile upłynęło klatek od uruchomienia programu

Na każdy punkt działa grawitacja równa 90*masa. Między punktami zachodzą siły sprężystości:

```
ofVec2f f = (dist - s->length) * KS + (vn * dpos) * KD / dist;
ofVec2f F = f * (dpos / dist);
p1->v_forces -= F;
p2->v_forces += F;
```

gdzie KS = 1755, KS = 35, **vn** jest róznicą prędkości punktów, **dist** jest skalarną odległością między punktami, **dpos** jest różnicą położeń punktów wyrażoną przez wektor, **s->length** jest długością liny łączącej oba punkty w położeniu równowagi.

Każdy punkt można „złapać” kursorem myszy (wtedy wskaźnik na kliknięty punkt zapisywany jest w odpowiedniej zmiennej **selectedPoint**, do której się odnoszę przy obliczeniach) i dodać mu siłę równą różnicy położeń między punktem a kursorem:

```
dragForce = (mousePos - selectedPoint->v_position);
selectedPoint->v_forces += dragForce;
```

Gracz:

Posiada wektory położenia i prędkości, swój rozmiar (promień), zmienną logiczną mówiącą nam, czy trzyma się jakiegoś punktu oraz wskaźnik na trzymany punkt.

W każdym kroku do prędkości gracza przypisywana jest siła grawitacji o wartości .5 (prędkość w y jest ograniczona do 15):

```
player->v_speed.y += .5; // grawitacja
if (player->v_speed.y >= 15) player->v_speed.y = 15; //ograniczenie prędkości spadania
```

Sprawdzana jest kolizja z podłożem:

```
// bardzo prowizoryczna kolizja z podłożem
if (player->v_position.y > groundHeight - player->m_radius)
{
    player->v_speed.y = 0;
    // a gdy się okaże, że wylądował pod podłogą
    if (player->v_position.y > groundHeight - player->m_radius)
        player->v_position.y = groundHeight - player->m_radius;
}
```

Skakanie odbywa się poprzez odjęcie od prędkości y wartości 15 tylko, gdy jest prędkość w y jest równa 0 oraz zachodzi kolizja z podłożem:

```
// bardzo prowizoryczne skakanie
if (player->v_position.y == groundHeight - player->m_radius) // gdy stoi na podłożu
    if (keyIsDown['w'] && player->v_speed.y == 0)
        player->v_speed.y = -15;
```

Poruszanie się poziomie polega na dodaniu lub odjęciu wartości 5 od prędkości w x. Dodałem również efekt „ślizgania” się.

Na koniec każdego kroku wartości prędkości gracza dodawane są do jego położenia.

```
// bardzo prowizoryczne sprawdzanie kolizji z punktami
if (keyIsDown['e'] && !player->isHoldingLine)
    for (auto &p : myPoints)
    {
        float dist = player->v_position.distance(p->v_position);
        if (dist < (player->m_radius + pointSize))
        {
            player->grabbedPoint = p;
            player->isHoldingLine = true;
            break;
        }
    }
```

<< Łapanie się punktów polega na sprawdzaniu kolizji z punktami gdy wciśnięty jest klawisz E. Wtedy adres do danego punktu przypisywany jest do wskaźnika **grabbedPoint** w obiekcie gracza a jego atrybut **isHoldingLine** zmieniany jest na *true*.

```
// bardzo prowizoryczne puszczanie się linki
if (player->isHoldingLine && keyIsDown['w'] && !keyIsDown['e'])
{
    player->grabbedPoint = nullptr;
    player->isHoldingLine = false;
    player->v_speed.y = -10;
}
```

<< Przy puszczaniu się linki (wciśnięcie klawisza W) zerowany jest wskaźnik **grabbedPoint**, **isHoldingLine** zmieniany jest na false oraz nadawana jest graczowi prędkość y = -10 .

```
// gdy gracz trzyma linkę jakomś
// bardzo prowizoryczne bujanie się na lince
if (player->isHoldingLine)
{
    player->grabbedPoint->v_forces.y += 500;
    if (keyIsDown['a'])
        player->grabbedPoint->v_forces.x -= 100;
    if (keyIsDown['d'])
        player->grabbedPoint->v_forces.x += 100;
}
```

<< Bujanie się na linie polega na dodawaniu 100 siły w odpowiednią stronę do złapanego przez gracza punktu.
Dodawana jest również siła w y, aby oddać wrażenie ciężkości gracza.

Podłoże ustalone jest jedynie przez zmienną przechowującą jego położenie w Y, w X rozciągnięte jest pod całej szerokości okna.

Napotkane problemy:

- ustalenie początkowych wartości prędkości punktów pod obliczenia metodą Verleta
- obliczenie prędkości przy użyciu metody Verleta
- „łapanie się” gracza za punkty
- system nasłuchiwania klawiszy

Rozwiązanie problemów:

Ustalenie początkowych wartości prędkości punktów pod obliczenia metodą Verleta.

Problem ten rozwiązałem opisanym już sposobem, przez obliczenie położeń w dwóch pierwszych krokach metodą Eulera. Dzięki temu mogłem zapełnić wektory **v_positionOld, v_position** oraz **v_positionNew** poprawnymi wartościami.

Obliczenie prędkości przy użyciu metody Verleta.

Rozwiązałem to licząc różnicę położeń między poprzednim a obecnym położeniem.

„Łapanie się” gracza za punkty.

Rozwiązałem to wykorzystując algorytm na kolizję dysków, dlatego też postanowiłem użyć dysku jako „modelu” gracza.

System nasłuchiwania klawiszy.

Niestety, sama metoda keyPressed zawarta już w OF była niewystarczająca, ponieważ wykrywała jedynie moment wciśnięcia klawisza. Potrzebowałem możliwości sprawdzania wciśnięcia wielu klawiszy na raz oraz sprawdzania czy dany klawisz jest wciśnięty przez dłuższy czas. Rozwiązałem to tworząc tablicę stanów **bool keyIsDown[255].** Jego zawartość modyfikowałem w metodach keyPressed oraz keyReleased. Dany stan zmieniany na true przy wciśnięciu danego klawisza, na false przy jego zwolnieniu:

```
void ofApp::keyPressed(int key){
    keyIsDown[key] = true;
}

//--------------------------------
void ofApp::keyReleased(int key){
    keyIsDown[key] = false;
}
```

Dzięki temu mam możliwość sprawdzania stanu klawisza również w pętli update(), na przykład przy wciśnięciu klawisza skoku:

```
if (keyIsDown['w'] && player->v_speed.y == 0)
    player->v_speed.y = -15;
```