SYMULATOR LECĄCYCH CZĄSTEK

Autor: Bartłomiej Damasiewicz

Akademia Górniczo-Hutnicza

Spis treści

1. Wstęp

Dokument dotyczy opracowania symulatora lecących cząstek. Symulator Lecących Cząstek to kompleksowe narzędzie stworzone z myślą o wizualizacji i badaniu dynamicznego ruchu cząstek w dwuwymiarowej przestrzeni. Głównym celem projektu jest dostarczenie użytkownikowi interaktywnego środowiska, które umożliwia eksperymentowanie z różnorodnymi scenariuszami oraz obserwowanie wpływu zmian parametrów na trajektorie i zachowanie się cząstek. Oferowane możliwości świetnie sprawdzą się w edukacyjnych, demonstracyjnych lub artystycznych. Zaimplementowany algorytm kolizji cząstek reprezentuje idealne zderzenia sprężyste.

Funkcje i możliwości:

1. **Wizualizacja ruchu cząstek:**

Symulator oferuje realistyczną wizualizację ruchu cząstek, pozwalając użytkownikowi na śledzenie ich trajektorii i pozycji w czasie rzeczywistym.

2. **Eksploracja scenariuszy:**

Użytkownik ma możliwość tworzenia i eksplorowania różnorodnych scenariuszy. Może manipulować parametrami cząstek, takimi jak rozmiar, kształt (kształt cząstki definiuje jej gęstość) oraz jej położenie, aby zobaczyć, jak zmiany te wpłyną na dynamikę układu.

3. **Wpływ na nauczanie i edukację**

Symulator doskonale sprawdza się jako narzędzie edukacyjne, umożliwiając nauczycielom i studentom zgłębianie wiedzy oraz intuicji związanych z fizyką ruchu.

4. **Zastosowanie artystyczne**

Pozwala na tworzenie abstrakcyjnych, dynamicznych kompozycji wizualnych, eksplorując estetykę ruchu cząstek wirtualnych.

Dokumentacja ma na celu dostarczenie pełnego zrozumienia możliwości, funkcji i potencjalnych zastosowań projektu. Zachęcam do eksperymentowania z symulatorem oraz wykorzystywania go w różnych kontekstach edukacyjnych.

2. Wymagania systemowe

1. **System operacyjny**
    - Windows
    - Linux
2. **Kompilator**
    - GCC
    - Visual C++
    - Clang
    - itp.
3. **Wersja języka oprogramowania**
    - Kompilator wspierający standard C++17
4. **Biblioteka OpenGL**
    - GLEW wersja 2.1.0
    - FreeGLUT wersja MSVC-3.0.0
5. **Rozdzielczość ekranu**
    - Minimalna rozdzielczość ekranu 1280x720
    - Zalecana rozdzielczość ekranu 1920x1080

6. **Karta graficzna**
    - Karta graficzna obsługująca OpenGL 2.0 lub nowsze

3. Funkcjonalność

1. **Interaktywny Interfejs użytkownika (Dear ImGui)**

Symulator posiada intuicyjny interfejs, który umożliwia użytkownikowi łatwe sterowanie parametrami symulacji, takimi jak wybór kształtu cząstki oraz rozmiar ich początkowe położenie. Interfejs umożliwia również zmianę koloru tła, resetowanie (usuwanie wszystkich cząstek znajdujących się na planszy) oraz wyświetlanie aktualnej liczby cząstek znajdujący się na planszy.

2. **Parametry Cząstek**

Użytkownik ma możliwość modyfikowania parametrów cząstek, takich jak rozmiar i kształt. Każdy kształt ma z góry określoną gęstość. Okrąg – 1, Sześciokąt – 2, Kwadrat – 3 oraz trójkąt – 4. Na podstawie rozmiaru oraz gęstości cząstki określana jest masa elementu. Kolor cząstek zmienia się płynnie od zielonego do czerwonego, określając czas życia danej cząstki.

3. **System Kolizji**

Symulator wykorzystuje algorytm kolizji, który bierze pod uwagę zarówno kolizje cząstek z otoczeniem, jak i wzajemne zderzenia między cząstkami.

4. **Wizualizację Trajektorii**

Użytkownik może śledzić trajektorię poruszających się cząstek, obserwując, jak zmiany w parametrach wpływają na ich ruch w czasie.

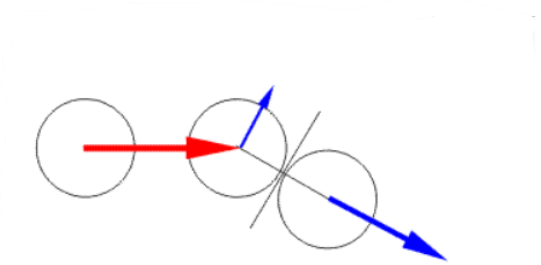
4. Analiza problemu

4.1 Cel Symulatora

Symulator ma na celu symulowanie ruchu cząstek w dwuwymiarowej przestrzeni. Mechanika zderzeń ma być reprezentowana w sposób sprężysty, co oznacza, że po zderzeniu cząstki powinny oddziaływać na siebie wzajemnie, zachowując energię kinetyczną.

4.2 Model Matematyczny

Zastosowanie matematycznego modelu opisującego ruch cząstek i ich zderzenia



$$v'_{1x} = \frac{v_1\cos(\theta_1-\varphi)\,(m_1-m_2)+2m_2v_2\cos(\theta_2-\varphi)}{m_1+m_2}\cos(\varphi) + v_1\sin(\theta_1-\varphi)\cos(\varphi+\frac{\pi}{2})$$

$$v'_y = \frac{v_1\cos(\theta_1-\varphi)\,(m_1-m_2)+2m_2v_2\cos(\theta_2-\varphi)}{m_1+m_2}\sin(\varphi) + v_1\sin(\theta_1-\varphi)\sin(\varphi+\frac{\pi}{2})$$

gdzie v_1, v_2 to skalarne wartości prędkości cząstek, m_1, m_2 wartości ich masy
θ_1, θ_2 są to kąty określające ich składowe części prędkości v_{1x}
$= v_1\cos(\theta_1)$ oraz φ to kąt kontaktu

Źródło: https://en.wikipedia.org/wiki/Elastic_collision#Two-dimensional

4.3 Idealne Zderzenia Sprężyste

Symulacja ma obejmować idealne zderzenia sprężyste, w których energii kinetyczna jest zachowana. Po zderzeniu cząstki powinny zmieniać swoje trajektorie i prędkości, ale suma ich energii kinetycznych pozostaje bez zmian.

4.4 Wpływ Parametrów na Zderzenia

Badanie wpływu parametrów, takich jak masa, rozmiar, gęstość cząstek na charakterystykę zderzeń. Umożliwienie użytkownikowi eksperymentowanie z różnymi scenariuszami, aby obserwować zmiany w ruchu cząstek.

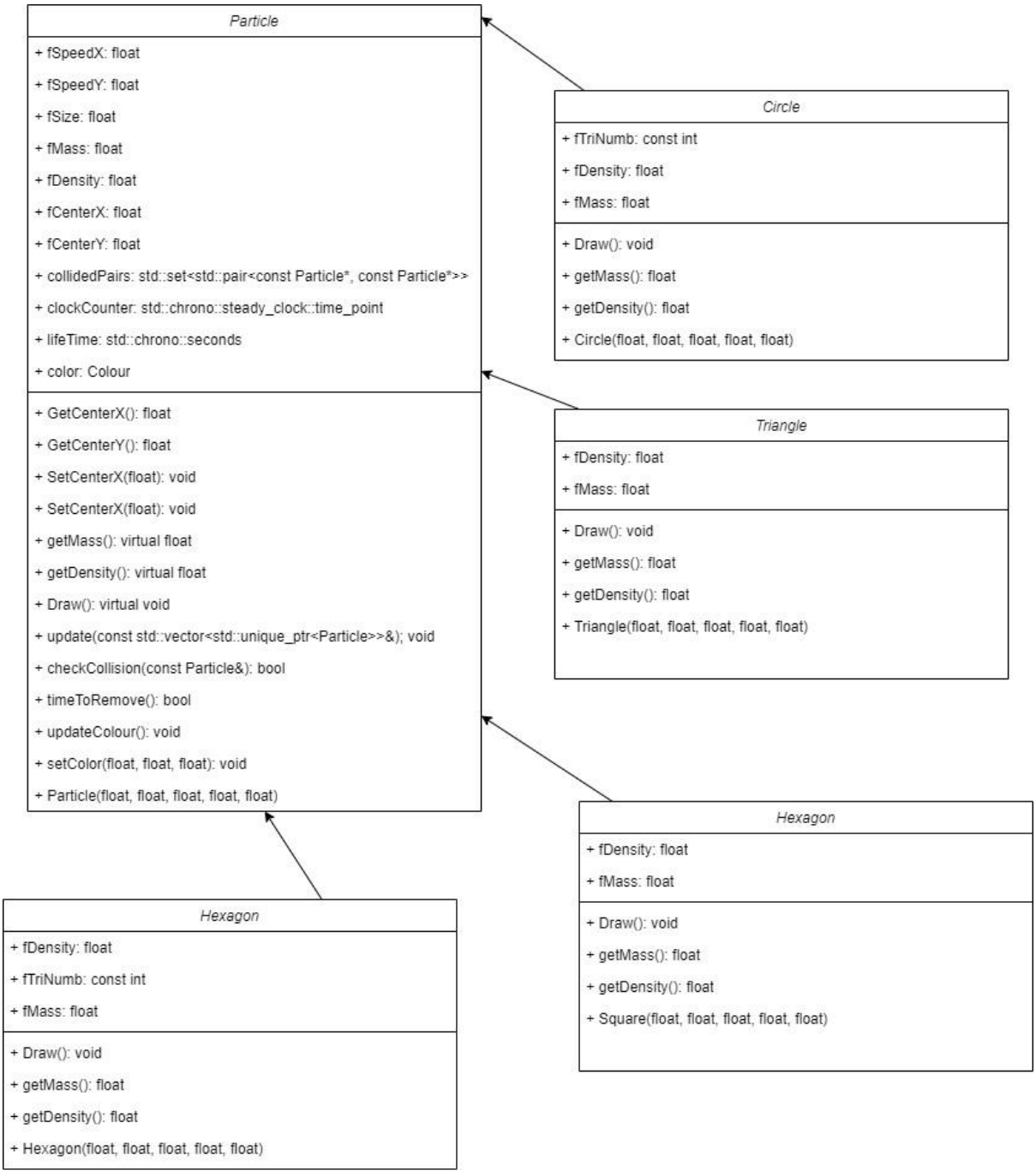
4.5 Algorytm Zderzeń

Zaimplementowany algorytm powinien skutecznie wykrywać i reagować na zderzenia między cząstkami. Uwzględniając kierunek i prędkość cząstek po zderzeniu.

4.6 Wizualizacja

Umożliwienie wizualizacji ruchu cząstek, ich zderzeń oraz ewentualnych zmian w kolorze czy innych atrybutach wizualnych w zależności od warunków symulacji.

5. Projekt techniczny



Rysunek 1. Przedstawia diagram UML w wersji uproszczonej

6. Opis realizacji

6.1.1 Opisy istotnych funkcji wraz z fragmentami kodu

Funkcja sprawdzająca wystąpienie kolizji.

Sprawdzanie kolizji opiera się na odległości między środkami dwóch cząstek oraz sumie ich promieni. Funkcja oblicza pierwiastek kwadratowy z kwadratu różnicy współrzędnych x i y środków dwóch cząstek. sumOfRad to suma promieni (rozmiarów). Funkcja zwraca „true”, jeśli odległość jest mniejsza lub równa sumie ich promieni.

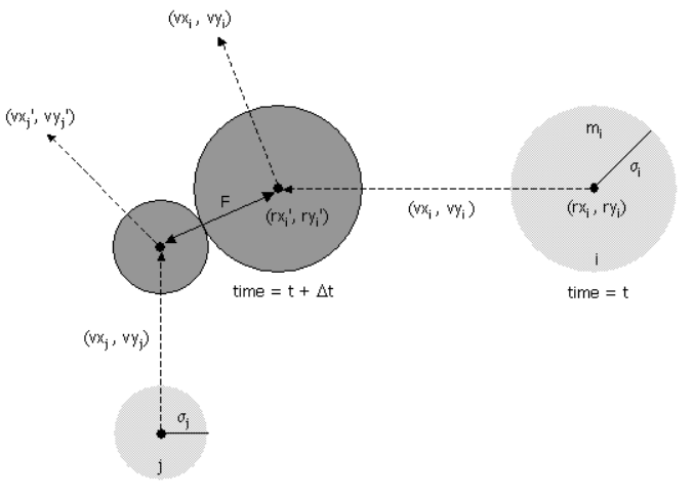
```
bool Particle::checkCollision(const Particle& otherParticle) const
{
    float distance =
        std::sqrt((fCenterX - otherParticle.fCenterX) * (fCenterX -
otherParticle.fCenterX) + (fCenterY - otherParticle.fCenterY) * (fCenterY -
otherParticle.fCenterY));
    float sumOfRad = fSize + otherParticle.fSize;
    return distance <= sumOfRad;
}
```

Funkcja aktualizująca pozycję cząstki.

Jest ona kluczową funkcją części symulacji ruchu cząstek w przestrzeni dwuwymiarowej.

Funkcja iteruje przez wszystkie elementy przekazane jako argument w wektorze „Particles” i sprawdza kolizje. Jeśli kolizja jest wykrywana, wykonuje się szereg obliczeń związanych z fizyką zderzeń, takich jak obliczania wektorów jednostkowych, impulsów i aktualizacja prędkości. Do obliczania prędkości po kolizji użyto algorytmu [1].

Opis algorytmu:



$$
\begin{aligned}
\Delta r &= (\Delta x,\ \Delta y) = (rx_j - rx_i,\ ry_j - ry_i)\\
\Delta v &= (\Delta vx,\ \Delta vy) = (vx_j - vx_i,\ vy_j - vy_i)\\
\Delta r \cdot \Delta r &= (\Delta x)^2 + (\Delta y)^2\\
\Delta v \cdot \Delta v &= (\Delta vx)^2 + (\Delta vy)^2\\
\Delta v \cdot \Delta r &= (\Delta vx)(\Delta x) + (\Delta vy)(\Delta y).
\end{aligned}
\qquad
J_x = \frac{J\,\Delta x}{\sigma},\quad J_y = \frac{J\,\Delta y}{\sigma},\quad \text{where}\quad J = \frac{2\,m_i\,m_j\,(\Delta v \cdot \Delta r)}{\sigma\,(m_i + m_j)}
$$

Poniżej przedstawiono fragment funkcji Particle::uptade(), która implementuje powyższy algorytm.

```
void Particle::update(const
std::vector<std::unique_ptr<Particle>>& Particles)
{
//…
if (distance < sumRadii) {
    // Obliczanie wektorów jednostkowych między
środkami obiektów
    float dx = otherParticle->fCenterX - fCenterX;
    float dy = otherParticle->fCenterY - fCenterY;
    float distanceNorm = std::sqrt(dx * dx + dy *
dy);

    if (distanceNorm > 0) {
        float nx = dx / distanceNorm;
        float ny = dy / distanceNorm;
        float overlap = sumRadii - distance;
        float pushX = (overlap * dx) / distanceNorm;
        float pushY = (overlap * dy) / distanceNorm;

        // Obliczanie prędkości względnej
        float relativeVelocityX = otherParticle-
>fSpeedX - fSpeedX;
        float relativeVelocityY = otherParticle-
>fSpeedY - fSpeedY;

        // Obliczanie iloczynu skalarnego prędkości
względnej i wektora normalnego
        float dotProduct = relativeVelocityX * nx +
relativeVelocityY * ny;

        // Obliczanie współczynnika impulsu
        float impulseCoeff = (2.0 * dotProduct) /
(getMass() + otherParticle-         >getMass());

        // Aktualizacja prędkości obiektów po
zderzeniu
        fSpeedX += impulseCoeff * otherParticle-
>getMass() * nx;
        fSpeedY += impulseCoeff * otherParticle-
>getMass() * ny;
        otherParticle->fSpeedX -= impulseCoeff *
getMass() * nx;
        otherParticle->fSpeedY -= impulseCoeff *
getMass() * ny;

//…
```

Funkcje do aktualizowania koloru cząstek

Funkcja setColor ustawia kolor cząstki na podstawie przekazanych wartości składowych koloru. Parametry „r", „g", „b" reprezentują składowe koloru w zakresie od 0.0 do 1.0. Funkcja przypisuje te wartości do struktury „Color".

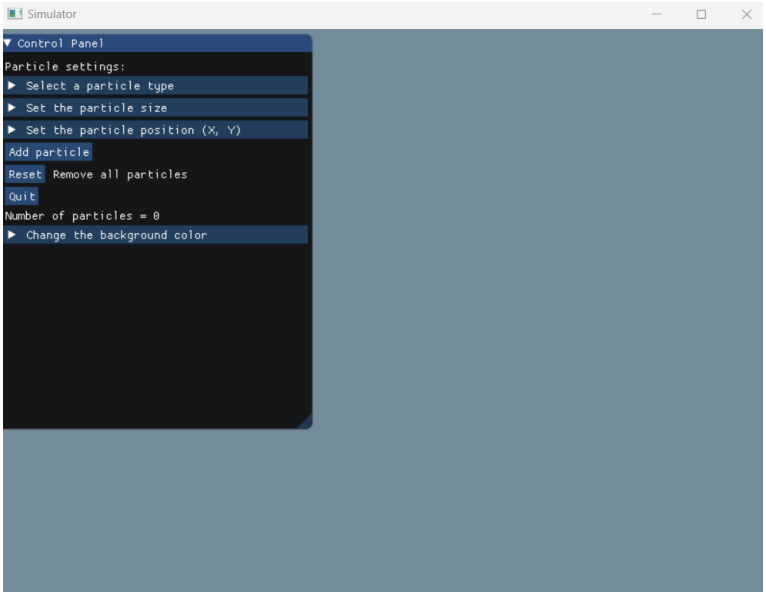
Funkcja updateColour aktualizuje kolor cząstki w zależności od czasu, który upłynął od momentu inicjalizacji cząstki (clockCounter). Aktualizacja koloru jest uzależniona od upływającego czasu, co może być używane do uzyskania efektu zmiany koloru w trakcie życia cząstki. W przykładowej implementacji, po upływie 40 sekund, kolor cząstki zmienia się z czerwonego na zielony.

```
void Particle::setColor(float r, float g, float b)
{
    color = {r, g, b};
}

void Particle::updateColour()
{
    auto currentTime = std::chrono::steady_clock::now();
    auto timeValue = std::chrono::duration_cast<std::chrono::seconds>(currentTime -
clockCounter).count();
    float newRed = 0.0f + static_cast<float>(timeValue/40.0);
    float newGreen = 1.0f - static_cast<float>(timeValue/40.0);
    float newBlue = 0.0f;

    setColor(newRed, newGreen, newBlue);
}
```

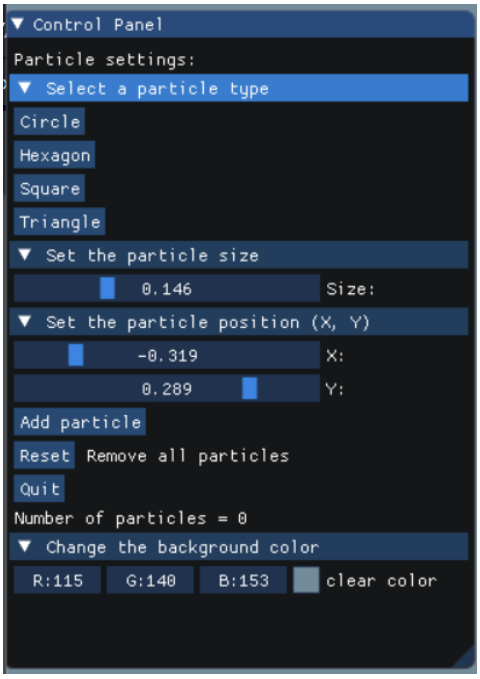
Do przedstawienia opisu realizacji użyto tylko kilka przykładowych funkcji. Reszta implementacji jest dostępna w kodzie źródłowym projektu.

7. Podręcznik użytkownika

Aby rozpocząć korzystanie z aplikacji należy uruchomić plik FPS_GLUT.exe w folderze FPS_GLUT\bin\Win32\Debug. Po uruchomieniu, użytkowni wyświetli się okno symulacji oraz menu wyboru.

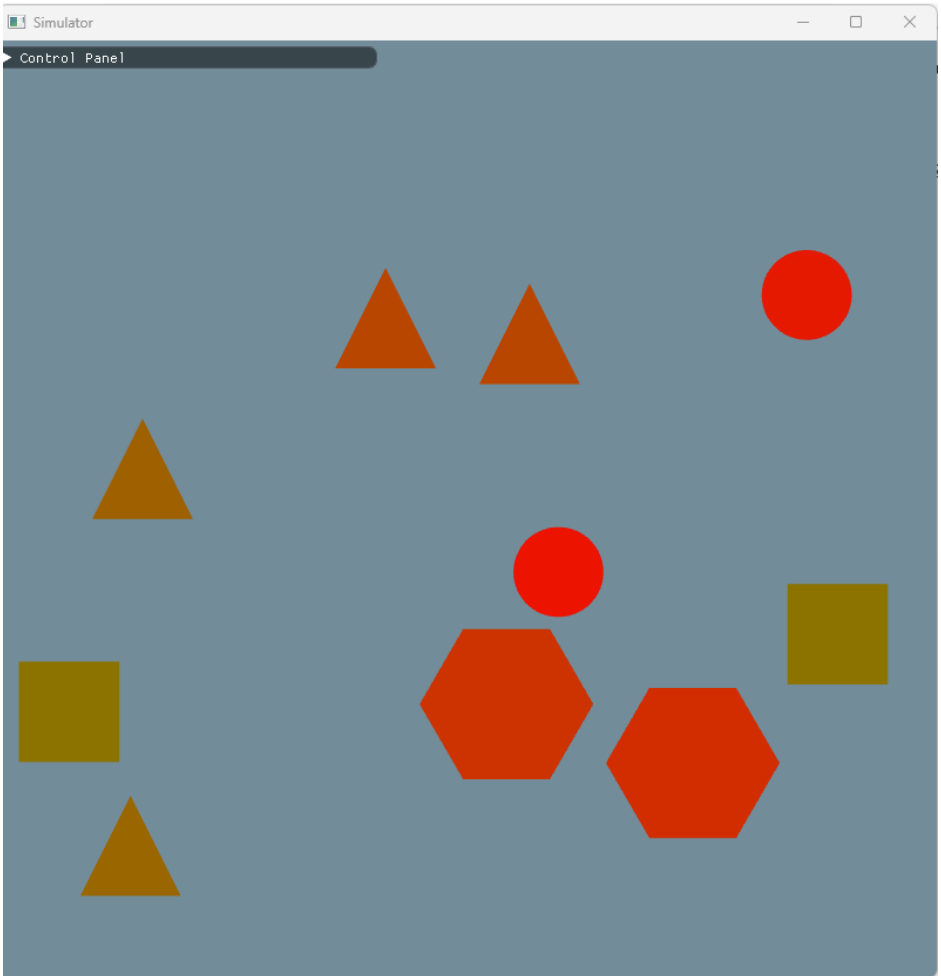


Następnie należy wybrac typ cząstki oraz jego parametry:



Aby cząstka pojawiła się na ekranie trzeba wybrać typ oraz określi jej rozmiar. Można także zmienić kolor tła symulacji naciskając przycisk po lewej stronie od “clear color”. Po wybraniu wszystkich parametrów należy nacisnąc przycisk “Add particle”. Pod przyciskiem “Quit” znajduje się licznik, który określa liczbę cząstek znajdujących się na planszy symulacji. Po zakończeniu symulacji można wyjść z programu przyciskiem “Quit”.

Poniżej przedsawiony jest przykład symulacji:



8. Metodologia rozwoju i utrzymania systemu

8.1.1 Iteracyjny Model Rozwoju:

Projekt był rozwijany przy użyciu iteracyjnego modelu rozwoju oprogramowania. Każda iteracja obejmował cykl planowania, projektowania, implementacji, testowania i oceny. Taki model umożliwił stopniowe wprowadzanie nowych funkcji, a także dostosowanie systemu do ewentualnych zmian wymagań.

8.1.2 System Kontroli Wersji:

Wszystkie pliki projektu, w tym kod źródłowy, dokumentację i zasoby graficzne, są przechowywane w systemie kontroli wersji, co ułatwia zarządzanie zmianami, śledzenie postępu pracy oraz przywracanie wcześniejszych wersji w przypadku konieczności.

8.1.3 Testowanie Jednostkowe i Testy Akceptacyjne:

Kod był poddawany systematycznym testom jednostkowym w celu weryfikacji poprawności implementacji poszczególnych funkcji.

8.1.4 Dokumentacja Kodu:

Kod źródłowy był odpowiednio komentowany, a kluczowe fragmenty są szczegółowo opisane w dokumentacji kodu.

8.1.5 Otwarty Dostęp do Kodu Źródłowego:

Kod źródłowy systemu będzie udostępniony publicznie na platformie GitHub, co pozwoli na partycypację społeczności, zgłaszanie błędów oraz zgłaszanie propozycji zmian.

Bibliografia

[1] https://introcs.cs.princeton.edu/java/assignments/collisions.html - algorytm kolizji cząstek. Ostatni dostęp 21.01.2024

[2] Cyganek B.: Programowanie w języku C++. Wprowadzenie dla inżynierów. PWN, 2023.

[3] https://freeglut.sourceforge.net/docs/api.php Ostatni dostęp 21.01.2024

[4] https://glew.sourceforge.net/basic.html Ostatni dostęp 21.01.2024

[5] https://github.com/ocornut/imgui/tree/92d29531fa5294e0684969f9dc33126379e6f02d - interfejs graficzny. Ostatni dostęp 21.01.2024