SPRAWOZDANIE

Zajęcia: Grafika komputerowa

Prowadzący: prof. dr hab. Vasyl Martenyuk

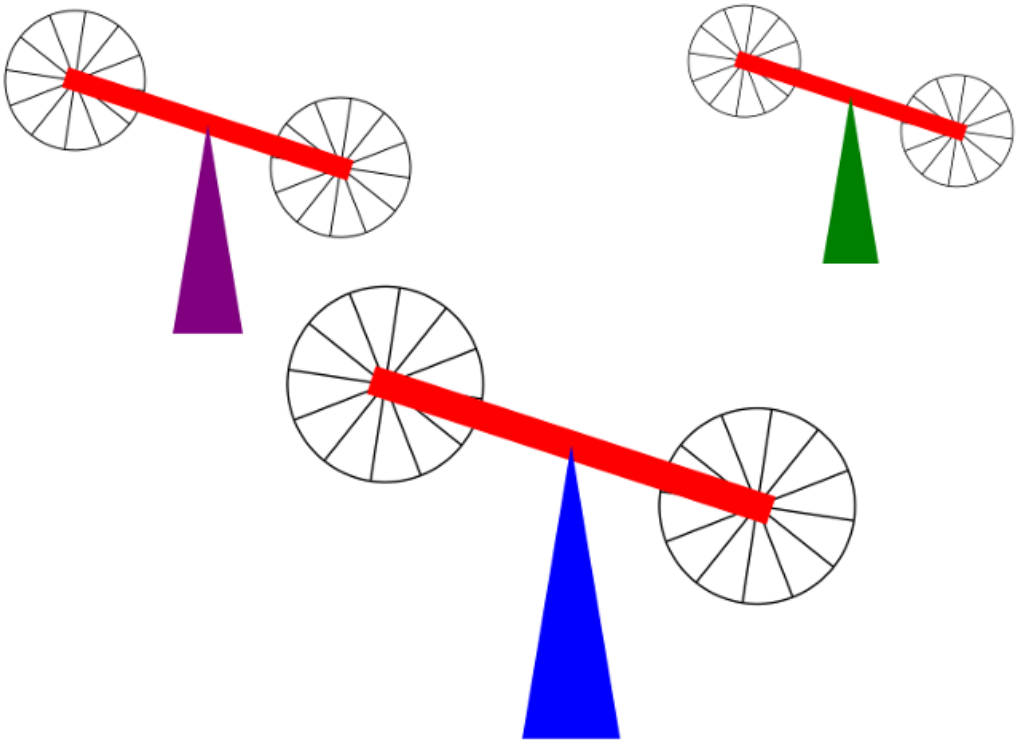
Laboratorium 3
21.03.2023
Temat: “Modelowanie hierarchiczne w grafice 2D”
Wariant: 15-kąt

Jakub Światłoch
Informatyka I stopień,
stacjonarne,
4 semestr,
Gr.2a

Pokazuję tylko kod który został zmodyfikowany, reszta kodu jest taka sama jaka została podana na zajęciach.

ZADANIE 1 a)

1. **Polecenie**: Stworzenie sceny hierarchicznej 2D z użyciem hierarchii funkcyjnej w JavaScript



Obraz nr 1

2. **Wprowadzane dane**: Tutaj opisuje moje własne stworzone funkcje

```
function drawHexagon(x, y, r) {
  var x0 = x;
  var y0 = y;
  graphics.beginPath();
  graphics.moveTo(x0 + r, y0);
  for (var i = 0; i <= 15; i++) {
    x = x0 + r * Math.cos((i * 2 * Math.PI) / 15);
    y = y0 + r * Math.sin((i * 2 * Math.PI) / 15);
    graphics.lineTo(x, y);
    graphics.moveTo(x0, y0);
    graphics.lineTo(x, y);
  }
  graphics.lineTo(x0 + r, y0);
  graphics.stroke();
}
```

drawHexagon() jest funkcją narysowania piętnastokąta na płaszczyźnie za pomocą elementów graficznych.

Argumenty funkcji to x, y i r, które reprezentują odpowiednio współrzędne poziome, pionowe oraz promień piętnastokąta.

Funkcja ta tworzy kontekst graficzny poprzez **graphics.beginPath()**, a następnie używa pętli for do obliczenia współrzędnych kolejnych punktów piętnastokąta i łączy je linią za pomocą **graphics.lineTo()**.

Ostatecznie funkcja zamyka kształt, łącząc ostatni punkt z pierwszym za pomocą **graphics.lineTo()**, i obramowuje go za pomocą **graphics.stroke()**.

Wywołanie funkcji **drawHexagon(0, 0, r)** rysuje piętnastokąt o promieniu r w punkcie (0,0) na płaszczyźnie.

```
function drawRectangle(x, y, g2, scale) {
  g2.save();
  g2.fillStyle = "red";
  g2.rotate((25 * Math.PI) / 60);
  g2.translate(x, y);
  g2.scale(0.25 * scale, 4 * scale);
  filledRect();
  g2.restore();
}
```

```
drawRectangle(-1, -0.2, graphics, 0.75);
```

drawRectangle() funkcją rysowania prostokąta na płaszczyźnie za pomocą elementów graficznych.

Argumenty funkcji to x, y, g2 i scale, które reprezentują odpowiednio współrzędne poziome, pionowe, kontekst graficzny i skalę rysowania.

Funkcja ta zapisuje stan kontekstu graficznego poprzez g2.save(), ustawia kolor wypełnienia **g2.fillStyle** na czerwony i obraca kontekst graficzny o 25 stopni za pomocą g2.rotate((25 * Math.PI) / 60).

Następnie funkcja przesuwa kontekst graficzny o zadane wartości współrzędnych x i y za pomocą **g2.translate(x, y)**. Skaluje również kontekst graficzny wzdłuż osi X i Y za pomocą **g2.scale(0.25 * scale, 4 * scale)**.

Po wykonaniu operacji na kontekście graficznym, funkcja rysuje prostokąt, korzystając z funkcji pomocniczej **filledRect()**. Ostatecznie funkcja przywraca początkowy stan kontekstu graficznego za pomocą **g2.restore()**.

```
function filledTriangleXY(x, y, g2, color, scale = 1) {
  g2.beginPath();
  g2.moveTo(x - 0.4 * scale, y);
  g2.lineTo(x + 0.4 * scale, y);
  g2.lineTo(x, y + 2.5 * scale);
  g2.closePath();
  g2.fillStyle = color;
  g2.fill();
}
```

```
filledTriangleXY(0, -2.9, graphics, "blue", 0.75);
```

Funkcja **filledTriangleXY()** rysuje wypełniony trójkąt na płaszczyźnie za pomocą elementów graficznych.

Argumenty funkcji to x, y, g2, color oraz scale, które reprezentują odpowiednio współrzędne poziome, pionowe, kontekst graficzny, kolor wypełnienia i skalę rysowania.

W funkcji najpierw definiowane są wierzchołki trójkąta za pomocą metody **moveTo** i **lineTo** na kontekście graficznym g2. Następnie trójkąt jest zamykany (wywołanie **closePath()**), aby stworzyć pełny kształt.

Po zdefiniowaniu kształtu, funkcja ustawia kolor wypełnienia trójkąta poprzez **g2.fillStyle = color** które podaje użytkownik w argumencie.

Na koniec funkcja rysuje wypełniony trójkąt korzystając z metody fill() na kontekście graficznym g2.

np wywołanie funkcji **filledTriangleXY(0, -2.9, graphics, "blue", 0.75)** rysuje trójkąt w kolorze niebieskim, przesunięty o (0, -2.9) względem punktu (0,0) na płaszczyźnie, o rozmiarze w skali 0.75. Wierzchołki trójkąta są ułożone w kształcie równoramiennego trójkąta, którego wierzchołek jest skierowany do góry.

```
function drawRotatingHexagon(x, y, r, scale) {
  graphics.save();
  graphics.translate(x, y);
  graphics.rotate((frameNumber * 0.75 * Math.PI) / 180);
  graphics.scale(scale, scale);
  drawHexagon(0, 0, r);
  graphics.restore();
}
```

```
drawRotatingHexagon(1.3, -1.35, 0.4, 1.5);
```

Funkcja **drawRotatingHexagon()** rysuje piętnastokąt foremny, który obraca się wokół własnej osi.

Argumenty funkcji to x, y, r i scale, które określają pozycję początkową, promień piętnastokąta, i skalę rysowania.

W funkcji najpierw wywołana jest metoda **save()** na kontekście graficznym graphics, co zapisuje jego aktualny stan. Następnie, po przesunięciu kontekstu do punktu (x, y) i skalowaniu, wywołana jest metoda **rotate()** z argumentem równym kątowi obrotu wyrażonemu w radianach. Kąt ten jest obliczany na podstawie zmiennej **frameNumber** i skali.

Po tym wszystkim wywoływana jest funkcja **drawHexagon**, która rysuje piętnastokąt o promieniu r i środku w punkcie (0, 0), położonym względem przesuniętego i obróconego kontekstu graficznego.

Ostatecznie funkcja przywraca kontekst graficzny do zapisanego wcześniej stanu za pomocą **restore()**.

Wywołanie funkcji **drawRotatingHexagon(1.3, -1.35, 0.4, 1.5)** rysuje obracający się piętnastokąt o środku w punkcie (1.3, -1.35), promieniu 0.4, i skali rysowania 1.5. Ruch piętnastokąta jest wyznaczany przez wartość zmiennej **frameNumber**, która jest licznikiem klatek animacji.

3. **Wykorzystane komendy wraz z opisem oraz wyniki działania**:

Użyte komendy zostały opisane krok wcześniej dlatego nie ma sensu tłumaczyć ich działania ponownie

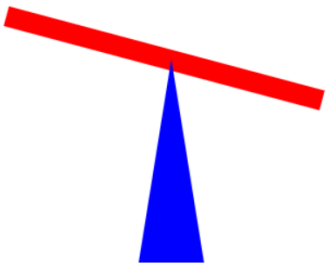
Rysowanie danego modelu krok po kroku:

Narysowanie samego trójkąta

```
function drawWorld() {
  filledTriangleXY(0, -2.9, graphics, "blue", 0.75);
```

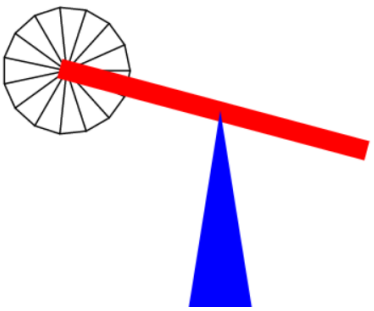
Narysowanie prostokąta do którego będą przyczepione piętnastokąty, **drawRectangle** trzeba dać nad **filledTrangleXY** aby był taki efekt jak na obrazie nr 1, czyli że prostokąta ma być za trójkątem

```
  graphics.restore();
}

function drawWorld() {
  drawRectangle(-1, -0.2, graphics, 0.75);
  filledTriangleXY(0, -2.9, graphics, "blue", 0.75);
}

function drawHexagon(x, y, r) {
  var x0 = x;
  var y0 = y;
```



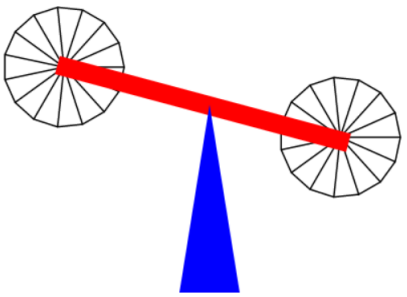
Podobnie z **drawRotatingHexagon**, trzeba go ustawić nad (za) prostokątem

```
  graphics.scale(scale, scale);
  drawHexagon(0, 0, r);
  graphics.restore();
}

function drawWorld() {
  drawRotatingHexagon(-1.45, -0.65, 0.4, 1.5);
  drawRectangle(-1, -0.2, graphics, 0.75);
  filledTriangleXY(0, -2.9, graphics, "blue", 0.75);
}

function drawHexagon(x, y, r) {
  var x0 = x;
```



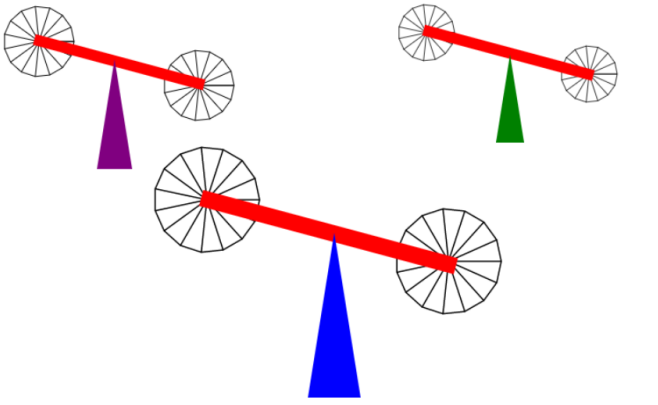
Podobnie będzie z piętnastokątem z prawej strony:

```
    drawHexagon(0, 0, r);
    graphics.restore();
  }

  function drawWorld() {
    drawRotatingHexagon(1.3, -1.35, 0.4, 1.5);
    drawRotatingHexagon(-1.45, -0.65, 0.4, 1.5);
    drawRectangle(-1, -0.2, graphics, 0.75);
    filledTriangleXY(0, -2.9, graphics, "blue", 0.75);
  }

  function drawHexagon(x, y, r) {
    var x0 = x;
```



Zrobienie pozostałych dwóch figur działa w taki sam sposób w jaki została wykonana powyższa figura, tylko wartości x, y, color i scale są odpowiednio dostosowane

```
    drawHexagon(0, 0, r);
    graphics.restore();
  }

  function drawWorld() {
    drawRotatingHexagon(1.3, -1.35, 0.4, 1.5);
    drawRotatingHexagon(-1.45, -0.65, 0.4, 1.5);
    drawRectangle(-1, -0.2, graphics, 0.75);
    filledTriangleXY(0, -2.9, graphics, "blue", 0.75);

    drawRotatingHexagon(-3.36, 1.15, 0.4, 1);
    drawRotatingHexagon(-1.54, 0.65, 0.4, 1);
    drawRectangle(0.25, 2.6, graphics, 0.5);
    filledTriangleXY(-2.5, -0.3, graphics, "purple", 0.5);

    drawRotatingHexagon(1.05, 1.25, 0.4, 0.8);
    drawRotatingHexagon(2.9, 0.78, 0.4, 0.8);
    drawRectangle(1.5, -1.65, graphics, 0.5);
    filledTriangleXY(2, 0, graphics, "green", 0.4);
  }
```



Link do GitHub w którym znajduje się kod z zadania
https://github.com/JakubSwiatlochgit/Grafika_Komputerowa/tree/main/cw3

Zadanie 2 b)

1. **Polecenie**: Stworzenie sceny hierarchicznej 2D z użyciem grafu sceny (podejście obiektowe) w JavaScript

2. **Wprowadzane dane**: W przypadku tego zadania stworzyłem swoje obiekty:

```
var hexagon = new SceneGraphNode();
hexagon.doDraw = function (g) {
  var vertices = 15;
  g.beginPath();
  for (var i = 0; i <= vertices; i++) {
    graphics.lineTo(
      1 * Math.cos((i * 2 * Math.PI) / vertices),
      1 * Math.sin((i * 2 * Math.PI) / vertices)
    );
    graphics.lineTo(0, 0);
    graphics.lineTo(
      1 * Math.cos((i * 2 * Math.PI) / vertices),
      1 * Math.sin((i * 2 * Math.PI) / vertices)
    );
  }
  g.closePath();
  g.stroke();
};
```

Ten kod trochę różni się od tego który tworzył piętnastokąt funkcyjnie ponieważ tworzenie obiektu do którego przesyłane byłyby parametry byłoby upierdliwe, dlatego lepiej zastosować taką prostą formę i w następnym etapie za pomocą transformacji wbudowanych przesuwać, skalować obiekt itp

Ten kod w JavaScript tworzy szablon piętnastokąta za pomocą funkcji "**doDraw**" obiektu "**hexagon**".

W kodzie zdefiniowano zmienną "**vertices**", która oznacza liczbę wierzchołków piętnastokąta . Następnie zdefiniowano pętlę "**for**", która iteruje po wierzchołkach piętnastokąta i rysuje linię łączącą wierzchołki za pomocą funkcji "**lineTo()**".

Wnętrze pętli składa się z trzech wywołań funkcji "lineTo()". Pierwsze i trzecie wywołanie rysuje linię z aktualnego punktu do kolejnego punktu na okręgu opisanym na piętnastokącie. Drugie wywołanie "**lineTo()**" rysuje linię z końcowego punktu do środka piętnastokącie.

Na końcu funkcja zamyka ścieżkę za pomocą funkcji "**closePath()**" i rysuje kontur piętnastokąt za pomocą "**stroke()**".

```
var triangle = new SceneGraphNode();
triangle.doDraw = function (g) {
  g.beginPath();
  g.moveTo(-0.5, 0);
  g.lineTo(0.5, 0);
  g.lineTo(0, 1);
  g.closePath();
  g.fill();
};
```

Ten kod w języku JavaScript tworzy szablon trójkąta za pomocą funkcji "**doDraw**" obiektu "**triangle**".

W kodzie zaczynamy od zdefiniowania funkcji "**beginPath()**", która rozpoczyna nową ścieżkę rysowania. Następnie definiujemy punkty, które tworzą trójkąt, korzystając z funkcji "**moveTo()**" oraz "**lineTo()**".

Pierwsze wywołanie funkcji "**moveTo()**" przesuwa punkt początkowy do pozycji (-0.5, 0), a kolejne wywołania funkcji "**lineTo()**" rysują linie z tego punktu do kolejnych punktów: (0.5, 0) oraz (0, 1), tworząc trójkąt.

Po zdefiniowaniu punktów, funkcja zamyka ścieżkę rysowania za pomocą funkcji "**closePath()**" i wypełnia obszar trójkąta kolorem wywołując funkcję "**fill()**".

3. **Wykorzystane komendy wraz z opisem**:

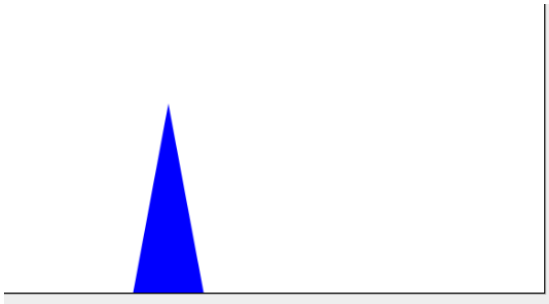
Aby rozpocząć zadanie trzeba zdefiniować pewne zmienne na których będzie się pracować:

```
let baseBlue;

let blueLeftHexagon;
let blueRightHexagon;
let blueTriangle;
```

Następnie zdefiniować te właśnie obiekty obiekty, np obiekt o nazwie "**baseBlue**". Obiekt ten powstaje w wyniku wywołania konstruktora klasy "**TransformedObject**" i przekazania do niego jako argumentu obiektu "**filledRect**".
Obiekt "**baseBlue**" jest więc utworzony na podstawie obiektu "**filledRect**" i ma wszystkie właściwości i metody zdefiniowane w klasie "**TransformedObject**". W związku z tym, że zmienna "**baseBlue**" jest używana do przechowywania obiektu, można ją traktować jako zmienną reprezentującą ten obiekt. Tak samo jest z pozostałymi obiektami

```
baseBlue = new TransformedObject(filledRect);

blueLeftHexagon = new TransformedObject(hexagon);
blueRightHexagon = new TransformedObject(hexagon);
blueTriangle = new TransformedObject(triangle);
```

Kod tworzy trójkąt o kolorze niebieskim, zmienia jego położenie, rozmiar oraz kolor, a następnie wyświetla go na ekranie.
Pierwsza linia kodu odnosi się do obiektu "**blueTriangle**", który już wcześniej został zdefiniowany w kodzie.
Następnie wywoływana jest na nim metoda "**setTranslation()**", która ustawia położenie trójkąta na scenie. W tym przypadku wartości (0, -3) oznaczają przesunięcie trójkąta o 0 jednostek w osi X i -3 jednostki w osi Y.
Kolejna metoda "**setColor()**" zmienia kolor trójkąta na niebieski.
Ostatnia metoda "**setScale()**" ustawia skalę trójkąta na (0.75, 2.0), co oznacza, że trójkąt będzie 0.75 raza mniejszy wzdłuż osi X i 2 razy większy wzdłuż osi Y.
Na końcu wywoływana jest metoda "**add()**" na obiekcie "**world**", która dodaje "**blueTriangle**" na ekran.



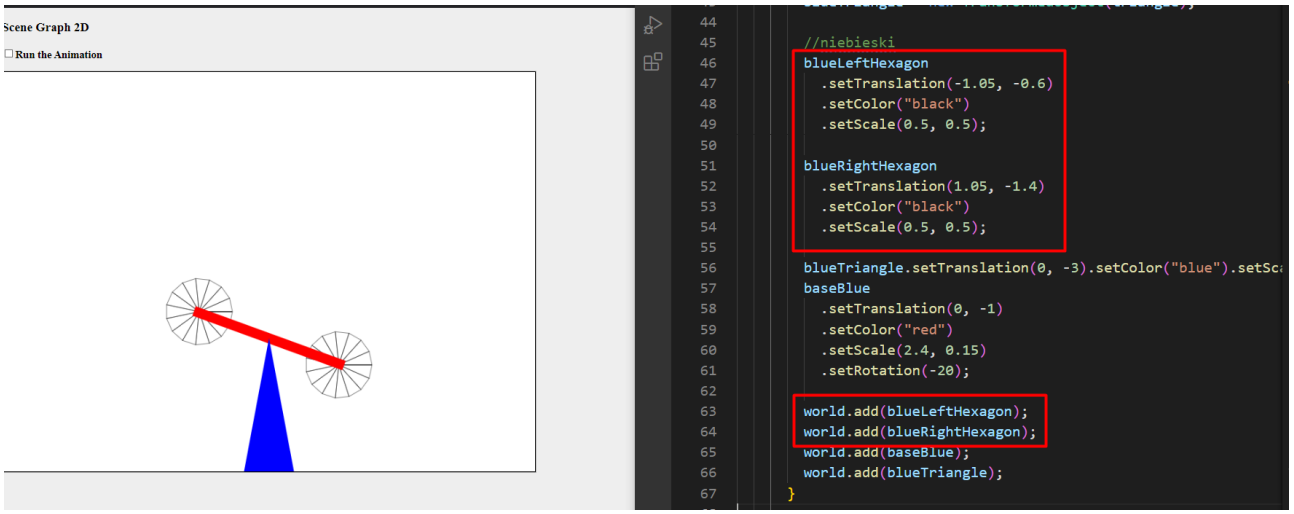
```
45      //niebieski
46
47      blueTriangle
48      .setTranslation(0, -3)
49      .setColor("blue")
50      .setScale(0.75, 2.0);
51
52      world.add(blueTriangle);
53  }
54
```

Ten kod w języku JavaScript definiuje niebieski prostokąt, który następnie jest przesuwany, skalowany, obracany i dodawany do świata gry.
Na początku kodu tworzony jest obiekt "**baseBlue**", który jest wcześniej zdefiniowany w kodzie.
Następnie wywołuje się na nim metodę "**setTranslation()**", która przesuwa pozycję prostokąta o (0, -1) jednostek w osi Y.
Kolejna metoda "**setColor()**" zmienia kolor prostokąta na czerwony.
Metoda "**setScale()**" zmienia skalę prostokąta na (2.4, 0.15), co oznacza, że prostokąt będzie 2.4 raza większy wzdłuż osi X i 0.15 raza mniejszy wzdłuż osi Y.
Ostatnia metoda "**setRotation()**" obraca prostokąt o -20 stopni (kąt ujemny oznacza obrót w lewo)
world.add dodaje **baseBlue** na ekran

Poniższy kod tworzy dwa piętnastokąty, jeden po lewej stronie i drugi po prawej stronie, na płótnie.

Piętnastokąty są koloru czarnego, mają skalę 0,5 i są przesunięte odpowiednio o -1,05 i -0,6 jednostek w przypadku lewego piętnastokąta oraz o 1,05 i -1,4 jednostek w przypadku prawego piętnastokąta .

Na koniec, piętnastokąty są dodawane do sceny (obiektu world).



Nie ma sensu tłumaczenia robienia pozostałych dwóch więc po prostu daje kod bez tłumaczeń

```
let baseBlue;
let basePurple;
let baseGreen;

let blueLeftHexagon;
let blueRightHexagon;
let blueTriangle;

let purpleLeftHexagon;
let purpleRightHexagon;
let purpleTriangle;

let greenLeftHexagon;
let greenRightHexagon;
let greenTriangle;
```

```
baseBlue = new TransformedObject(filledRect);
basePurple = new TransformedObject(filledRect);
baseGreen = new TransformedObject(filledRect);

blueLeftHexagon = new TransformedObject(hexagon);
blueRightHexagon = new TransformedObject(hexagon);
blueTriangle = new TransformedObject(triangle);

purpleLeftHexagon = new TransformedObject(hexagon);
purpleRightHexagon = new TransformedObject(hexagon);
purpleTriangle = new TransformedObject(triangle);

greenLeftHexagon = new TransformedObject(hexagon);
greenRightHexagon = new TransformedObject(hexagon);
greenTriangle = new TransformedObject(triangle);
```

```
//fioletowy

purpleLeftHexagon
  .setTranslation(-3.04, 1.89)
  .setColor("black")
  .setScale(0.4, 0.4);

purpleRightHexagon
  .setTranslation(-0.95, 1.12)
  .setColor("black")
  .setScale(0.4, 0.4);

purpleTriangle
  .setTranslation(-2.0, -0.1)
  .setColor("purple")
  .setScale(0.4, 1.6);

basePurple
  .setTranslation(-2, 1.5)
  .setColor("red")
  .setScale(2.4, 0.15)
  .setRotation(-20);
```

```
//green

greenLeftHexagon
  .setTranslation(1.35, 2.25)
  .setColor("black")
  .setScale(0.3, 0.3);

greenRightHexagon
  .setTranslation(2.65, 1.76)
  .setColor("black")
  .setScale(0.3, 0.3);

greenTriangle
  .setTranslation(2.0, 1)
  .setColor("green")
  .setScale(0.25, 1);

baseGreen
  .setTranslation(2, 2)
  .setColor("red")
  .setScale(1.5, 0.1)
  .setRotation(-20);
```

```
world.add(purpleLeftHexagon);
world.add(purpleRightHexagon);
world.add(basePurple);
world.add(purpleTriangle);

world.add(greenLeftHexagon);
world.add(greenRightHexagon);
world.add(baseGreen);
world.add(greenTriangle);
```

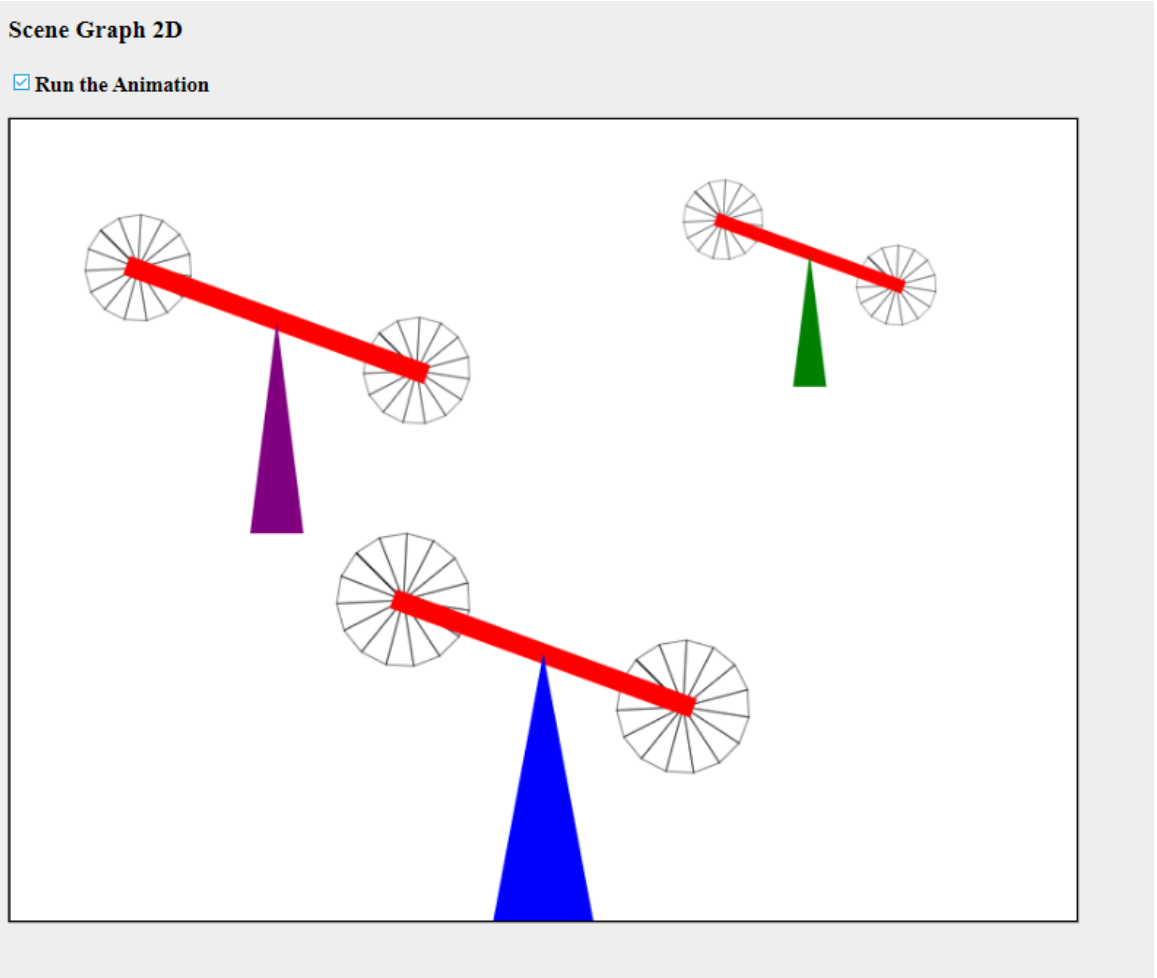
Jeszcze aby animacje działały po kliknięciu w przycisk trzeba dodać poniższe wartości w updateFrame

```
function updateFrame() {
  frameNumber++;

  // TODO: Update state in preparation for drawing the

  blueLeftHexagon.setRotation(frameNumber * 15);
  blueRightHexagon.setRotation(frameNumber * 15);

  purpleLeftHexagon.setRotation(frameNumber * 15);
  purpleRightHexagon.setRotation(frameNumber * 15);

  greenLeftHexagon.setRotation(frameNumber * 15);
  greenRightHexagon.setRotation(frameNumber * 15);
}
```

Ten kod ustawia rotację (obrót) piętnastokąta nazwaHexagon w każdej klatce animacji na wartość równą **frameNumber** * 15.

frameNumber to zmienna przechowująca aktualny numer klatki animacji. Mnożenie przez 15 oznacza, że piętnastokąt będzie obracał się z prędkością 15 stopni na klatkę animacji. W ten sposób, przy każdej kolejnej klatce, piętnastokąt będzie obracany o 15 stopni w stosunku do poprzedniej pozycji.

4. Wyniki



Wnioski:
(a) Zalety używania hierarchii funkcji:

-Ułatwia to organizowanie kodu w sposób modularny i uporządkowany.
-Pozwala na łatwe wielokrotne wykorzystanie kodu.
-Umożliwia zastosowanie zasad DRY (Don't Repeat Yourself) - unikanie powtarzania kodu.
-Dzięki wykorzystaniu parametrów funkcji, można łatwo dostosować ich działanie do różnych potrzeb.

Wady hierarchii funkcji:

-Przy zbyt dużym zagłębieniu funkcji może być trudno zrozumieć działanie kodu.
-Używanie zbyt dużej liczby funkcji może wpłynąć na wydajność programu.

(b) Zalety tworzenia grafu sceny:

-Umożliwia łatwe i przejrzyste zarządzanie obiektami na scenie.
-Pozwala na łatwe skalowanie i modyfikowanie sceny.
-Umożliwia łatwe dodawanie i usuwanie obiektów na scenie.

-Dzięki zastosowaniu dziedziczenia można tworzyć złożone hierarchie obiektów.

Wady tworzenia grafu sceny:

-Może wymagać więcej czasu na naukę i zrozumienie działania programu.
-Zbyt duża liczba obiektów i złożoność hierarchii może wpłynąć na wydajność programu.

Wnioski:
Obie techniki mają swoje zalety i wady. Decyzja o wyborze jednej z nich zależy od indywidualnych potrzeb i wymagań projektu. Hierarchia funkcji jest bardziej użyteczna dla prostych i małych projektów, gdzie obiektowość może wprowadzić zbyt dużą złożoność. Graf sceny natomiast jest bardziej skomplikowany, ale pozwala na łatwe zarządzanie obiektami na scenie i jest bardziej użyteczny dla większych i bardziej złożonych projektów.