

Uniwersytet Bielsko-Bialski

Obraz zawierający tekst, Czcionka, logo, Grafika

Opis wygenerowany automatycznie

***Grafika Komputerowa***

# (ćwiczenia laboratoryjne)

**Laboratoria numer: 3.**

# Temat ćwiczenia: Modulowanie hierarchiczne w grafice 2D.

Data wykonania ćwiczenia: 13.04.2023

Wykonała:

Krzysztof Wierzbicki

# Cel zadania

1. Opracować scenę hierarchiczną zgodnie z obrazem używając zamiast kół wielokąty obracające się (animacja!) według wariantu. Opracowanie powinno być w jednym z języków: Java lub JavaScript,
2. na dwa sposoby:
3. (a) używając hierarchiję funkcje (sposób subroutinowy)
4. (b) tworząc graf sceny (sposób obiektowy). W tym celu proponuję do pobrania odpowiedni pliki

# 2. Wyniki działania

Scene Graph 2D

Obraz zawierający tekst, zrzut ekranu, diagram, linia

Opis wygenerowany automatycznie

<!DOCTYPE html>

<html>

    <head>

        <meta charset="UTF-8" />

        <title>Scene Graph 2D</title>

        <script>

            var canvas; // The canvas that is used as the drawing surface

            var graphics; // The 2D graphics context for drawing on the canvas.

            var X\_LEFT = -4; // The xy limits for the coordinate system.

            var X\_RIGHT = 4;

            var Y\_BOTTOM = -3;

            var Y\_TOP = 3;

            var BACKGROUND = "white"; // The display is filled with this color before the scene is drawn.

            var pixelSize; // The size of one pixel, in the transformed coordinates.

            var frameNumber = 0; // Current frame number. goes up by one in each frame.

            var world; // A SceneGraphNode representing the entire scene.

            // TODO: Define global variables to represent animated objects in the scene.

            var rotatingFan1;

            var rotatingFan2;

            var rotatingFan3;

            /\*\*

             \*  Builds the data structure that represents the entire picture.

             \*/

            function createWorld() {

                world = new CompoundObject(); // Root node for the scene graph.

                rotatingFan1 = new TransformedObject(rotatingFan1);

                rotatingFan1.setScale(1, 1).setColor("red").setTranslation(-0.5, -0.5);

                rotatingFan2 = new TransformedObject(rotatingFan2);

                rotatingFan2.setScale(0.5, 0.5).setColor("green").setTranslation(2, 2);

                rotatingFan3 = new TransformedObject(rotatingFan3);

                rotatingFan3.setScale(0.75, 0.75).setColor("blue").setTranslation(-3, 2);

                world.add(rotatingFan1);

                world.add(rotatingFan2);

                world.add(rotatingFan3);

            }

            /\*\*

             \* This method is called just before each frame is drawn.  It updates the modeling

             \* transformations of the objects in the scene that are animated.

             \*/

            function updateFrame() {

                frameNumber++;

            }

            //------------------- A Simple Scene Object-Oriented Scene Graph API ----------------

            /\*\*

             \* The (abstract) base class for all nodes in the scene graph data structure.

             \*/

            function SceneGraphNode() {

                this.fillColor = null; // If non-null, the default fillStyle for this node.

                this.strokeColor = null; // If non-null, the default strokeStyle for this node.

            }

            SceneGraphNode.prototype.doDraw = function (g) {

                // This method is meant to be abstract and must be OVERRIDDEN in an actual

                // object. It is not meant to be called; it is called by draw().

                throw "doDraw not implemented in SceneGraphNode";

            };

            SceneGraphNode.prototype.draw = function (g) {

                // This method should be CALLED to draw the object It should NOT

                // ordinarily be overridden in subclasses.

                graphics.save();

                if (this.fillColor) {

                    g.fillStyle = this.fillColor;

                }

                if (this.strokeColor) {

                    g.strokeStyle = this.strokeColor;

                }

                this.doDraw(g);

                graphics.restore();

            };

            SceneGraphNode.prototype.setFillColor = function (color) {

                // Sets fillColor for this node to color.

                // Color should be a legal CSS color string, or null.

                this.fillColor = color;

                return this;

            };

            SceneGraphNode.prototype.setStrokeColor = function (color) {

                // Sets strokeColor for this node to color.

                // Color should be a legal CSS color string, or null.

                this.strokeColor = color;

                return this;

            };

            SceneGraphNode.prototype.setColor = function (color) {

                // Sets both the fillColor and strokeColor to color.

                // Color should be a legal CSS color string, or null.

                this.fillColor = color;

                this.strokeColor = color;

                return this;

            };

            /\*\*

             \*  Defines a subclass, CompoundObject, of SceneGraphNode to represent

             \*  an object that is made up of sub-objects.  Initially, there are no

             \*  sub-objects.  Objects are added with the add() method.

             \*/

            function CompoundObject() {

                SceneGraphNode.call(this); // do superclass initialization

                this.subobjects = []; // the list of sub-objects of this object

            }

            CompoundObject.prototype = new SceneGraphNode(); // (makes it a subclass!)

            CompoundObject.prototype.add = function (node) {

                this.subobjects.push(node);

                return this;

            };

            CompoundObject.prototype.doDraw = function (g) {

                // Just call the sub-objects' draw() methods.

                for (var i = 0; i < this.subobjects.length; i++)

                    this.subobjects[i].draw(g);

            };

            /\*\*

             \*  Define a subclass, TransformedObject, of SceneGraphNode that

             \*  represents an object along with a modeling transformation to

             \*  be applied to that object.  The object must be specified in

             \*  the constructor.  The transformation is specified by calling

             \*  the setScale(), setRotate() and setTranslate() methods. Note that

             \*  each of these methods returns a reference to the TransformedObject

             \*  as its return value, to allow for chaining of method calls.

             \*  The modeling transformations are always applied to the object

             \*  in the order scale, then rotate, then translate.

             \*/

            function TransformedObject(object) {

                SceneGraphNode.call(this); // do superclass initialization

                this.object = object;

                this.rotationInDegrees = 0;

                this.scaleX = 1;

                this.scaleY = 1;

                this.translateX = 0;

                this.translateY = 0;

            }

            TransformedObject.prototype = new SceneGraphNode(); // (makes it a subclass!)

            TransformedObject.prototype.setRotation = function (angle) {

                // Set the angle of rotation, measured in DEGREES.  The rotation

                // is always about the origin.

                this.rotationInDegrees = angle;

                return this;

            };

            TransformedObject.prototype.setScale = function (sx, sy) {

                // Sets scaling factors.

                this.scaleX = sx;

                this.scaleY = sy;

                return this;

            };

            TransformedObject.prototype.setTranslation = function (dx, dy) {

                // Set translation mounts.

                this.translateX = dx;

                this.translateY = dy;

                return this;

            };

            TransformedObject.prototype.doDraw = function (g) {

                // Draws the object, with its modeling transformation.

                g.save();

                if (this.translateX != 0 || this.translateY != 0) {

                    g.translate(this.translateX, this.translateY);

                }

                if (this.rotationInDegrees != 0) {

                    g.rotate((this.rotationInDegrees / 180) \* Math.PI);

                }

                if (this.scaleX != 1 || this.scaleY != 1) {

                    g.scale(this.scaleX, this.scaleY);

                }

                this.object.draw(g);

                g.restore();

            };

            // Create some basic shapes as custom SceneGraphNode objects.

            const drawFan = (graphics) => {

                drawHexagon(graphics, 0.5, -0.4, 0.39);

                drawHexagon(graphics, 0.5, 2.42, -0.35);

                drawFilledRectTilted(graphics, 3, 0.2, 1, 0, -15);

                filledTriangle(graphics, 0.5, 2, 1, -1);

            };

            var rotatingFan1 = new SceneGraphNode();

            rotatingFan1.doDraw = drawFan;

            var rotatingFan2 = new SceneGraphNode();

            rotatingFan2.doDraw = drawFan;

            var rotatingFan3 = new SceneGraphNode();

            rotatingFan3.doDraw = drawFan;

            // ------------------------------- graphics support functions --------------------------

            /\*\*

             \* Draw one frame of the animation.  Probably doesn't need to be changed,

             \* except maybe to change the setting of preserveAspect in applyLimits().

             \*/

            function draw() {

                graphics.save(); // to make sure changes don't carry over from one call to the next

                graphics.fillStyle = BACKGROUND; // background color

                graphics.fillRect(0, 0, canvas.width, canvas.height);

                graphics.fillStyle = "black";

                applyLimits(graphics, X\_LEFT, X\_RIGHT, Y\_TOP, Y\_BOTTOM, false);

                graphics.lineWidth = pixelSize; // Use 1 pixel as the default line width

                world.draw(graphics);

                graphics.restore();

            }

            function filledTriangle(g2, width, height, centerX, centerY) {

                g2.beginPath();

                g2.moveTo(centerX, centerY + height / 2);

                g2.lineTo(centerX + width / 2, centerY - height / 2);

                g2.lineTo(centerX - width / 2, centerY - height / 2);

                g2.closePath();

                g2.fill();

            }

            function drawHexagon(g2, size, centerX, centerY) {

    const numSides = 16; // Zmieniamy ilość boków na 16

    const rotationAngle = ((frameNumber % 360) \* Math.PI) / 180;

    g2.save();

    g2.translate(centerX, centerY);

    g2.rotate(rotationAngle);

    g2.translate(-centerX, -centerY);

    g2.beginPath();

    for (let i = 0; i < numSides; i++) {

        const angleRad = (Math.PI / (numSides / 2)) \* i; // Podział pełnego obrót na 16 części

        const x = centerX + size \* Math.cos(angleRad);

        const y = centerY + size \* Math.sin(angleRad);

        if (i === 0) {

            g2.moveTo(x, y);

        } else {

            g2.lineTo(x, y);

        }

    }

    g2.closePath();

    g2.stroke();

    g2.beginPath();

    for (let i = 0; i < numSides; i++) {

        const angleRad = (Math.PI / (numSides / 2)) \* i; // Podział pełnego obrót na 16 części

        const x = centerX + size \* Math.cos(angleRad);

        const y = centerY + size \* Math.sin(angleRad);

        g2.moveTo(centerX, centerY);

        g2.lineTo(x, y);

    }

    g2.stroke();

    g2.restore();

}

            function drawFilledRectTilted(

                g2,

                width,

                height,

                centerX,

                centerY,

                angle

            ) {

                g2.save();

                g2.translate(centerX, centerY);

                g2.rotate((angle \* Math.PI) / 180);

                g2.fillRect(-width / 2, -height / 2, width, height);

                g2.restore();

            }

            /\*\*

             \* Applies a coordinate transformation to the graphics context, to map

             \* xleft,xright,ytop,ybottom to the edges of the canvas.  This is called

             \* by draw().  This does not need to be changed.

             \*/

            function applyLimits(

                g,

                xleft,

                xright,

                ytop,

                ybottom,

                preserveAspect

            ) {

                var width = canvas.width; // The width of this drawing area, in pixels.

                var height = canvas.height; // The height of this drawing area, in pixels.

                if (preserveAspect) {

                    // Adjust the limits to match the aspect ratio of the drawing area.

                    var displayAspect = Math.abs(height / width);

                    var requestedAspect = Math.abs(

                        (ybottom - ytop) / (xright - xleft)

                    );

                    var excess;

                    if (displayAspect > requestedAspect) {

                        excess =

                            (ybottom - ytop) \*

                            (displayAspect / requestedAspect - 1);

                        ybottom += excess / 2;

                        ytop -= excess / 2;

                    } else if (displayAspect < requestedAspect) {

                        excess =

                            (xright - xleft) \*

                            (requestedAspect / displayAspect - 1);

                        xright += excess / 2;

                        xleft -= excess / 2;

                    }

                }

                var pixelWidth = Math.abs((xright - xleft) / width);

                var pixelHeight = Math.abs((ybottom - ytop) / height);

                pixelSize = Math.min(pixelWidth, pixelHeight);

                g.scale(width / (xright - xleft), height / (ybottom - ytop));

                g.translate(-xleft, -ytop);

            }

            //------------------ Animation framework ------------------------------

            var running = false; // This is set to true when animation is running

            function frame() {

                if (running) {

                    // Draw one frame of the animation, and schedule the next frame.

                    updateFrame();

                    draw();

                    requestAnimationFrame(frame);

                }

            }

            function doAnimationCheckbox() {

                var shouldRun = document.getElementById("animateCheck").checked;

                if (shouldRun != running) {

                    running = shouldRun;

                    if (running) requestAnimationFrame(frame);

                }

            }

            //----------------------- initialization -------------------------------

            function init() {

                canvas = document.getElementById("thecanvas");

                if (!canvas.getContext) {

                    document.getElementById("message").innerHTML =

                        "ERROR: Canvas not supported";

                    return;

                }

                graphics = canvas.getContext("2d");

                document.getElementById("animateCheck").checked = false;

                document.getElementById("animateCheck").onchange =

                    doAnimationCheckbox;

                createWorld();

                draw();

            }

        </script>

    </head>

    <body onload="init()" style="background-color: #eeeeee">

        <h3>Scene Graph 2D</h3>

        <noscript>

            <p>

                <b style="color: red"

                    >Error: This page requires JavaScript, but it is not

                    available.</b

                >

            </p>

        </noscript>

        <p id="message">

            <label

                ><input type="checkbox" id="animateCheck" /><b

                    >Run the Animation</b

                ></label

            >

        </p>

        <div style="float: left; border: 2px solid black">

            <canvas

                id="thecanvas"

                width="800"

                height="600"

                style="display: block"

            ></canvas>

        </div>

    </body>

</html>

Obraz zawierający zrzut ekranu, diagram, tekst, linia

Opis wygenerowany automatycznie

<!DOCTYPE html>

<html>

    <head>

        <meta charset="UTF-8" />

        <title>Subroutine Hieararchy</title>

        <script>

            var canvas; // The canvas that is used as the drawing surface

            var graphics; // The 2D graphics context for drawing on the canvas.

            var X\_LEFT = -4; // The xy limits for the coordinate system.

            var X\_RIGHT = 4;

            var Y\_BOTTOM = -3;

            var Y\_TOP = 3;

            var BACKGROUND = "white"; // The display is filled with this color before the scene is drawn.

            var pixelSize; // The size of one pixel, in the transformed coordinates.

            var frameNumber = 0; // Current frame number. goes up by one in each frame.

            // TODO:  Define any other necessary state variables.

            /\*\*

             \*  Responsible for drawing the entire scene.  The display is filled with the background

             \*  color before this function is called.

             \*/

            function drawWorld() {

                drawRoot(1, 0.75, 1, "blue");

                drawRoot(3, -2, 0.75, "purple");

                drawRoot(-2, -2, 0.5, "green");

            }

            /\*\*

             \* This method is called just before each frame is drawn.  It updates the modeling

             \* transformations of the objects in the scene that are animated.

             \*/

            function updateFrame() {

                frameNumber++;

                // TODO: If other updates are needed for the next frame, do them here.

            }

            function drawRoot(centerX, centerY, scale, color) {

                drawHexagon(

                    graphics,

                    0.5 \* scale,

                    -0.4 \* scale - centerX,

                    0.39 \* scale - centerY

                );

                drawHexagon(

                    graphics,

                    0.5 \* scale,

                    2.42 \* scale - centerX,

                    -0.35 \* scale - centerY

                );

                graphics.fillStyle = "red";

                drawFilledRectTilted(

                    graphics,

                    3 \* scale,

                    0.2 \* scale,

                    1 \* scale - centerX,

                    0 \* scale - centerY,

                    -15

                );

                graphics.fillStyle = color;

                filledTriangle(

                    graphics,

                    0.5 \* scale,

                    2 \* scale,

                    1 \* scale - centerX,

                    -1 \* scale - centerY

                );

            }

            //------------------- Some methods for drawing basic shapes. ----------------

            function line() {

                // Draws a line from (-0.5,0) to (0.5,0)

                graphics.beginPath();

                graphics.moveTo(-0.5, 0);

                graphics.lineTo(0.5, 0);

                graphics.stroke();

            }

            function rect() {

                // Strokes a square, size = 1, center = (0,0)

                graphics.strokeRect(-0.5, -0.5, 1, 1);

            }

            function filledRect() {

                // Fills a square, size = 1, center = (0,0)

                graphics.fillRect(-0.5, -0.5, 1, 1);

            }

            function circle() {

                // Strokes a circle, diameter = 1, center = (0,0)

                graphics.beginPath();

                graphics.arc(0, 0, 0.5, 0, 2 \* Math.PI);

                graphics.stroke();

            }

            function filledCircle() {

                // Fills a circle, diameter = 1, center = (0,0)

                graphics.beginPath();

                graphics.arc(0, 0, 0.5, 0, 2 \* Math.PI);

                graphics.fill();

            }

            function filledTriangle(g2, width, height, centerX, centerY) {

                g2.beginPath();

                g2.moveTo(centerX, centerY + height / 2);

                g2.lineTo(centerX + width / 2, centerY - height / 2);

                g2.lineTo(centerX - width / 2, centerY - height / 2);

                g2.closePath();

                g2.fill();

            }

            function drawFilledRectTilted(

                g2,

                width,

                height,

                centerX,

                centerY,

                angle

            ) {

                g2.save();

                g2.translate(centerX, centerY);

                g2.rotate((angle \* Math.PI) / 180);

                g2.fillRect(-width / 2, -height / 2, width, height);

                g2.restore();

            }

            function drawHexagon(g2, size, centerX, centerY) {

    const rotationAngle = ((frameNumber % 360) \* Math.PI) / 180;

    g2.save();

    g2.translate(centerX, centerY);

    g2.rotate(rotationAngle);

    g2.translate(-centerX, -centerY);

    g2.beginPath();

    for (let i = 0; i < 16; i++) { // zmiana z 6 na 16

        const angleRad = (Math.PI / 8) \* i; // obliczanie kąta dla szesnastokąta

        const x = centerX + size \* Math.cos(angleRad);

        const y = centerY + size \* Math.sin(angleRad);

        if (i === 0) {

            g2.moveTo(x, y);

        } else {

            g2.lineTo(x, y);

        }

    }

    g2.closePath();

    g2.stroke();

    g2.beginPath();

    for (let i = 0; i < 16; i++) { // zmiana z 6 na 16

        const angleRad = (Math.PI / 8) \* i; // obliczanie kąta dla szesnastokąta

        const x = centerX + size \* Math.cos(angleRad);

        const y = centerY + size \* Math.sin(angleRad);

        g2.moveTo(centerX, centerY);

        g2.lineTo(x, y);

    }

    g2.stroke();

    g2.restore();

}

            // ------------------------------- graphics support functions --------------------------

            /\*\*

             \* Draw one frame of the animation.  Probably doesn't need to be changed,

             \* except maybe to change the setting of preserveAspect in applyLimits().

             \*/

            function draw() {

                graphics.save(); // to make sure changes don't carry over from one call to the next

                graphics.fillStyle = BACKGROUND; // background color

                graphics.fillRect(0, 0, canvas.width, canvas.height);

                graphics.fillStyle = "black";

                applyLimits(graphics, X\_LEFT, X\_RIGHT, Y\_TOP, Y\_BOTTOM, false);

                graphics.lineWidth = pixelSize; // Use 1 pixel as the default line width

                drawWorld();

                graphics.restore();

            }

            /\*\*

             \* Applies a coordinate transformation to the graphics context, to map

             \* xleft,xright,ytop,ybottom to the edges of the canvas.  This is called

             \* by draw().  This does not need to be changed.

             \*/

            function applyLimits(

                g,

                xleft,

                xright,

                ytop,

                ybottom,

                preserveAspect

            ) {

                var width = canvas.width; // The width of this drawing area, in pixels.

                var height = canvas.height; // The height of this drawing area, in pixels.

                if (preserveAspect) {

                    // Adjust the limits to match the aspect ratio of the drawing area.

                    var displayAspect = Math.abs(height / width);

                    var requestedAspect = Math.abs(

                        (ybottom - ytop) / (xright - xleft)

                    );

                    var excess;

                    if (displayAspect > requestedAspect) {

                        excess =

                            (ybottom - ytop) \*

                            (displayAspect / requestedAspect - 1);

                        ybottom += excess / 2;

                        ytop -= excess / 2;

                    } else if (displayAspect < requestedAspect) {

                        excess =

                            (xright - xleft) \*

                            (requestedAspect / displayAspect - 1);

                        xright += excess / 2;

                        xleft -= excess / 2;

                    }

                }

                var pixelWidth = Math.abs((xright - xleft) / width);

                var pixelHeight = Math.abs((ybottom - ytop) / height);

                pixelSize = Math.min(pixelWidth, pixelHeight);

                g.scale(width / (xright - xleft), height / (ybottom - ytop));

                g.translate(-xleft, -ytop);

            }

            //------------------ Animation framework ------------------------------

            var running = false; // This is set to true when animation is running

            function frame() {

                if (running) {

                    // Draw one frame of the animation, and schedule the next frame.

                    updateFrame();

                    draw();

                    requestAnimationFrame(frame);

                }

            }

            function doAnimationCheckbox() {

                var shouldRun = document.getElementById("animateCheck").checked;

                if (shouldRun != running) {

                    running = shouldRun;

                    if (running) requestAnimationFrame(frame);

                }

            }

            //----------------------- initialization -------------------------------

            function init() {

                canvas = document.getElementById("thecanvas");

                if (!canvas.getContext) {

                    document.getElementById("message").innerHTML =

                        "ERROR: Canvas not supported";

                    return;

                }

                graphics = canvas.getContext("2d");

                document.getElementById("animateCheck").checked = false;

                document.getElementById("animateCheck").onchange =

                    doAnimationCheckbox;

                draw();

            }

        </script>

    </head>

    <body onload="init()" style="background-color: #eeeeee">

        <h3>Subroutine Hierarchy</h3>

        <noscript>

            <p>

                <b style="color: red">

                    Error: This page requires JavaScript, but it is not

                    available.

                </b>

            </p>

        </noscript>

        <p id="message">

            <label>

                <input type="checkbox" id="animateCheck" />

                <b>Run the Animation</b>

            </label>

        </p>

        <div style="float: left; border: 2px solid black">

            <canvas

                id="thecanvas"

                width="800"

                height="600"

                style="display: block"

            ></canvas>

        </div>

    </body>

</html>

# 3. Wnioski

Ćwiczenie pozwala zrozumieć strukturę danych potrzebną do hierarchicznego modelowania w grafice 2D, co jest kluczowe dla organizacji informacji na różnych poziomach szczegółowości.

Hierarchiczna struktura ułatwia zarządzanie złożonymi projektami graficznymi oraz umożliwia ponowne wykorzystanie komponentów, co przyczynia się do efektywności pracy.

Praktyka manipulacji obiektami w hierarchicznej strukturze pomaga opanować techniki, takie jak translacja, rotacja i skalowanie, co przekłada się na różnorodność efektów w grafice 2D.

Zrozumienie hierarchicznego modelowania może być przydatne w projektowaniu interaktywnych aplikacji 2D, gdzie elementy interfejsu użytkownika są dynamicznie manipulowane w zależności od działań użytkownika.

Hierarchiczne modelowanie odgrywa kluczową rolę w animacji 2D, umożliwiając płynne ruchy postaci i obiektów, co przekłada się na lepsze doświadczenia wizualne dla użytkowników.

Świadomość wpływu hierarchii na wydajność renderowania jest istotna dla projektowania efektywnych systemów graficznych, podkreślając znaczenie optymalizacji i wydajności.