

# JavaFX 3D

lebo 2D Canvas už každý videl

Lukáš Gajdošech

Programovanie(4)

2. mája 2020

- 1 Základy (3D v JavaFX, pozícia kamery, group/pane)
- 2 Kamera (projekcie, depth buffer, generovanie prostredia)
- 3 Objekty (správa objektov, aktualizovanie pozície, mazanie)
- 4 Vylepšenia (materiály, svetlá, transformácie, UI)
- 5 Modely a GUI (import, vlastné modely, UI panely)
- 6 Interakcia (pohyb hráča, rotácie, kolízie, raycast, ukážky)


```
public class Sphere3D extends Application {  
  
    @Override  
    public void start(Stage primaryStage) {  
        Sphere sphere = new Sphere(50);  
  
        Group group = new Group();  
        group.getChildren().add(sphere);  
  
        Camera camera = new PerspectiveCamera();  
        Scene scene = new Scene(group, 1280, 720);  
        scene.setCamera(camera);  
  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
}
```

# Základný kód

Zatiaľ nemusíte veriť, že to je naozaj 3D, no vyzerá to tak:

```
@SuppressWarnings("Duplicates")
public class Sphere3D extends Application {


    @Override
    public void start() {
        Sphere sphere = new Sphere(100, 100, 100, Color.BLUE);
        Group group = new Group();
        group.getChildren().add(sphere);
        Scene scene = new Scene(1000, 1000);
        // ak by sme chceli použiť kameru
        Camera camera = new Camera(1000, 1000, 1000, 1000);
        scene.setCamera(camera);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

A small window showing a 3D rendering of a sphere. The sphere is light gray and is positioned in the center of the window. The window has a standard macOS-style title bar with red, yellow, and green buttons.

Ak by sme namiesto Sphere použili 2D Circle (Circle2D.java):

```
@SuppressWarnings("Duplicates")
public class Circle2DGroup extends Application {

    @Override
    public void start() {
        Circle circle = new Circle(100, 100, 100, Color.BLUE);
        Group group = new Group();
        group.getChildren().add(circle);
        Scene scene = new Scene(1000, 1000);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

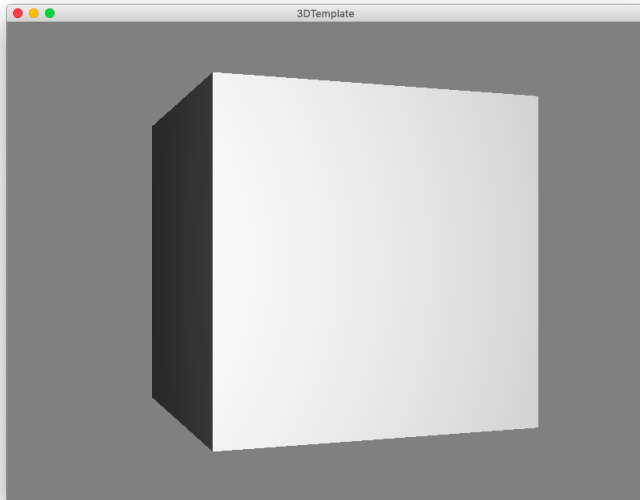
A small window showing a 2D rendering of a circle. The circle is light gray and is positioned in the center of the window. The window has a standard macOS-style title bar with red, yellow, and green buttons.

- Namiesto Circle používame Sphere.
- V 2D sa pozeráme na plochu, ktorá sa rozširuje v dvoch smeroch.
- V 3D už potrebujeme **kameru**, ktorá určí, odkiaľ sa na svet pozeráme. Preto vytvárame PerspectiveCamera, ktorú priradíme scéne.
- Prečo sa ale sféra zobrazila v ľavom hornom rohu? V 2D bol toto počiatok systému súradníc 0, 0. V 3D je to počiatok 0, 0, 0.
- Ak sa nám teda z pohľadu kamery zdá, že je sféra v ľavom hornom rohu, aká je potom poloha kamery vo svete a kto ju za nás nastavil?

- Kamera je implicitne nastavená tak, aby videla počiatok sústavy v ľavom hornom rohu.
- V skutočnosti ju ani vôbec nemusíme vytvárať a JavaFX si ju vytvorí automaticky.
- JavaFX je preto v podstate **vždy 3D**. Pri 2D sa pozeráme na rovnú plochu, ktorú máme rovno pred očami.
- Dôsledkom toho môžeme aj pri tvorbe 3D aplikácie používať 2D objekty, ako kruh či štvorec. Normálne sa zobrazia v scéne (ako plochy bez hĺbky) a dokonca im môžeme meniť aj Z súradnicu.

- V praxi vytvoríme kameru radšej ako `new PerspectiveCamera(true)`; pričom bool parameter zaručuje, že sa už kamera v 3D svete správa tak, ako by sme čakali, čiže sa vytvorí v počiatku súradnicového systému.
- Následne môžeme meniť jej súradnice a pohybovať sa po svete.
- Kamere vieme nastaviť aj `setFarClip(int a)`; a `setNearClip(int a)`; . To ovplyvňuje do akej vzdialenosti kamera dovidí. Ďalej ako `FarClip`, respektíve bližšie ako `NearClip` sa už objekty nerenderujú.
- Túto situáciu možno vidieť v `CameraAtOrigin.java`. To je odporúčaný template pre začiatok ľubovoľného 3D projektu.

# Template



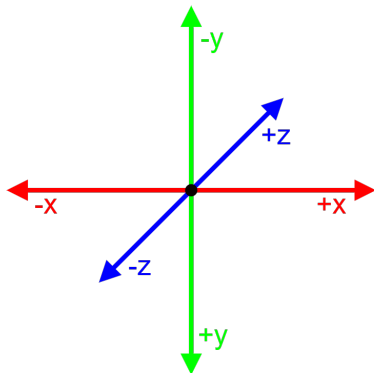


# A čo je ten Group?

- V kódoch používam Group namiesto Pane, na ktorý ste zvyknutí z 2D.
- Keď sa pozrieme do hierarchie tried JavaFX, Group je vyššie ako Pane, ide o všeobecnejšiu triedu s menej funkciami.
- Pane má nastaviteľnú veľkosť a podtypy ako BorderPane, ktoré umožňujú vytvárať špecifický layout aplikácie.
- Group iba obaluje skupinu JavaFX objektov (Nodes), slúži ako kontajner.
- Pane má preto zmysel pri 2D, kedy slúži ako plátno s vlastnou veľkosťou a layoutom. V 3D to nemá zmysel, preto stačí Group, no všetky kódy by fungovali aj s Pane...

# Súradnicový systém

- Pravotočivá sústava, i keď sa na ňu pozeráme trochu z inej strany, než je zvykom.
- **Diagram** je väčšinou odpoveďou na všetky nevysvetliteľné problémy...



- 1 Základy (3D v JavaFX, pozícia kamery, group/pane)
- 2 Kamera (projekcie, depth buffer, generovanie prostredia)
- 3 Objekty (správa objektov, aktualizovanie pozície, mazanie)
- 4 Vylepšenia (materiály, svetlá, transformácie, UI)
- 5 Modely a GUI (import, vlastné modely, UI panely)
- 6 Interakcia (pohyb hráča, rotácie, kolízie, raycast, ukážky)

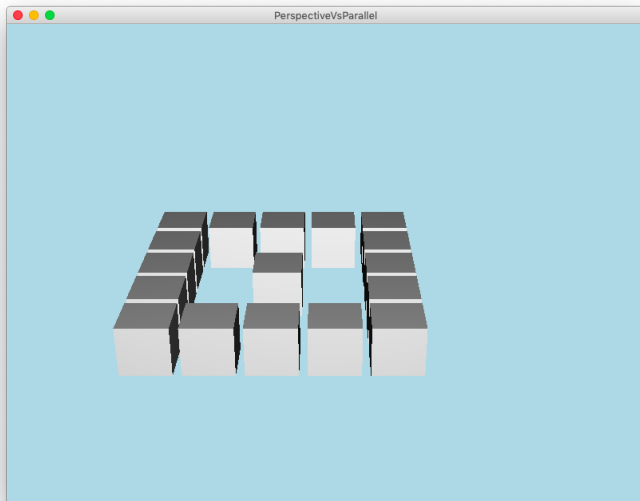
- Okrem PerspectiveCamera máme aj ParallelCamera.
- V grafike je toto typ projekcie, pričom projekcia je proces mapovania 3D sveta na 2D plochu (monitor).
- Paralelná projekcia je jednoduchšia, zanedbáva vzdialenosť objektov od kamery, čím sa stráca perspektíva.
- Vo všeobecnosti je pre nás prirodzenejšia perspektívna projekcia, no paralelná môže mať zmysel pre umelecký dojem (izometrické hry).
- **Ukážka:** majme aplikáciu, ktorá vytvorí mestečko kociek zo zadanej dvojrozmernej mapy.
- Na začiatku tvorby takýchto aplikácií si treba **rozmyslieť**, kde budú pre nás svetové strany a kde nebo/zem.
- V nasledujúcej ukážke uvažujeme, že zem je v smere osi Y.

# Ukážka projekcií

```
public class PerspectiveVsParallel extends Application {  
    ...  
    private char[][] map = {  
        {'#', '#', '#', '#', '#'},  
        {'#', '.', '.', '.', '#'},  
        {'#', '.', '#', '.', '#'},  
        {'#', '.', '.', '.', '#'},  
        {'#', '#', '#', '#', '#'},  
    };  
    private void loadMap() {  
        int Z = 0;  
        for (int i = map.length - 1; i >= 0; i--) {  
            int X = 0;  
            for (int j = 0; j < map[0].length; j++) {  
                if (map[i][j] == '#') createBox(X, Z);  
                X += 70;  
            }  
            Z += 70;  
        }  
    }  
}
```

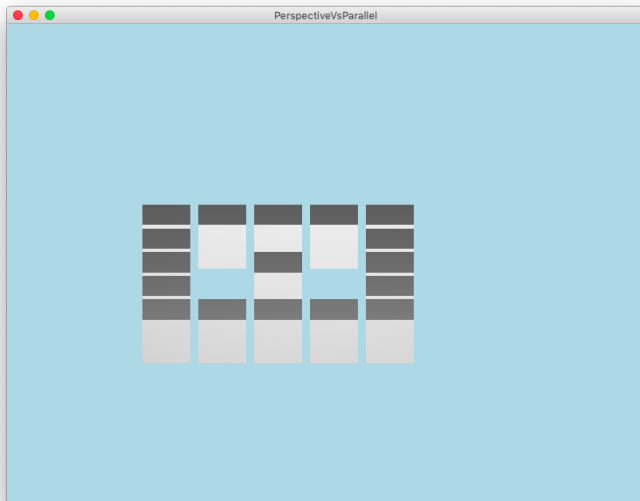
# Perspektívna projekcia

```
Camera camera = new PerspectiveCamera();
```



# Paralelná projekcia

```
Camera camera = new ParallelCamera();
```



# Umelecký dojem...



Obr.: Ukážka z hry Pillars of Eternity II: Deadfire

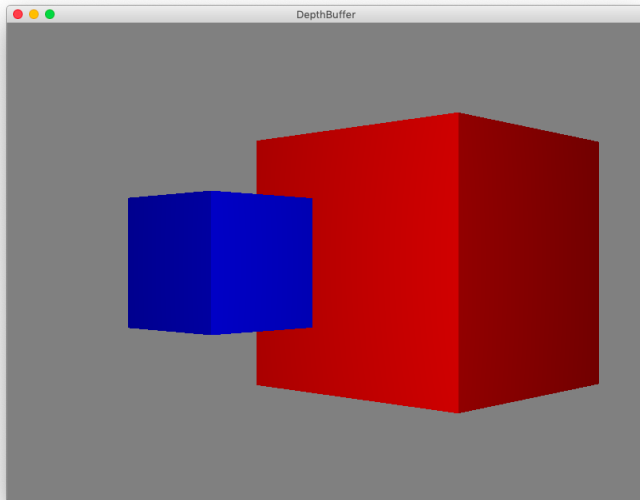


- V 3D sa objekty môžu prekryvať, vďaka čomu niektoré objekty nemusíme vidieť.
- Na určenie toho, ktorý objekt sa má **vyrenderovať** a ktorý nie slúži takzvaný depth buffer.
- To je výsledok výpočtu, ktorý určí vzdialenosti objektov od kamery. Ak sa nejaké prekryvajú, systém potom vykreslí iba ten, ktorý je ku kamere bližšie.
- V JavaFX je výpočet depth bufferu defaultne vypnutý. Výpočet totiž musí prebiehať každý frame a nie vždy ho potrebujeme. Väčšina 3D aplikácií ho však vyžaduje.
- Zapneme ho posledným nepovinným bool argumentom v konštrukture scény `new Scene(group, WIDTH, HEIGHT, true);`

# Aj Depth Buffer je len obrázok...



# Význam Depth Buffera



- 1 Základy (3D v JavaFX, pozícia kamery, group/pane)
- 2 Kamera (projekcie, depth buffer, generovanie prostredia)
- 3 Objekty (správa objektov, aktualizovanie pozície, mazanie)**
- 4 Vylepšenia (materiály, svetlá, transformácie, UI)
- 5 Modely a GUI (import, vlastné modely, UI panely)
- 6 Interakcia (pohyb hráča, rotácie, kolízie, raycast, ukážky)

# Odbočka: Dynamické objekty

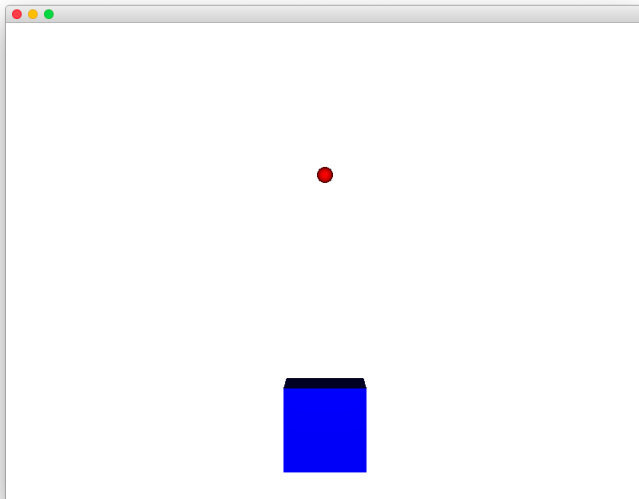
- Nasledujúca úvaha platí rovnako aj v prípade 2D.
- Predstavte si aplikáciu s objektami, ktoré každý frame menia polohu, prípadne sa aj dynamicky objavujú/miznú. Príklady: guľka skákaľka, veľa guliek skákaliek, projektily zo zbrane, lietajúce motýle...
- Ako to implementujeme? Ktorý spôsob je **najoptimálnejší**?
- V nasledujúcich ukážkach sú objekty aktualizované každých 20milisekúnd (čiže 50 krát za sekundu), pomocou Timeline:

```
Timeline animation = new Timeline(  
    new KeyFrame(Duration.millis(20), e -> update()));  
animation.setCycleCount(Timeline.INDEFINITE);  
animation.play();
```

- Vystrelenie projektilu je riadené udalosťou z klávesnice:

```
if (event.getCode().equals(KeyCode.SPACE)  
    && projectileY == DEFAULT_Y) {  
    projectileY -= 5;  
}
```

# Očakávaný výsledok



# Spôsob 1


Pamätáme si iba súradnice objektov, vytvárame ich v každom frame nanovo.

```
public class Projectiles1 extends Application {
    private final int DEFAULT_Y = HEIGHT - HEIGHT / 6;
    private int projectileY = DEFAULT_Y;
    ...
    private void update() {
        if (projectileY < -20) projectileY = DEFAULT_Y;
        group.getChildren().clear();
        Box player = new Box(100, 100, 100); ...
        if (projectileY != DEFAULT_Y) {
            Sphere projectile = new Sphere(10);
            projectile.setTranslateX(WIDTH / 2);
            projectile.setTranslateY(projectileY);
            group.getChildren().add(projectile);
            projectileY -= 5;
        }
    }
}
```

# Spôsob 1

Pamätáme si iba súradnice objektov, vytvárame ich v každom frame nanovo.

```
public class Projectiles1 extends Application {  
    private final int DEFAULT_Y = HEIGHT - HEIGHT/6;  
    private int  
    ...  
    private void  
        if (pro  
            group.g  
            Box play  
            if (pro  
                Sph  
                pro  
                projectile.setTranslateY(projectileY);  
                group.getChildren().add(projectile);  
                projectileY -= 5;  
        }  
    }  
}
```





## Spôsob 2


Objekty si vytvoríme na začiatku a potom im iba meníme súradnice a pridávame ich do detí skupiny.

```
public class Projectiles2 extends Application {  
    private Box player = new Box(100, 100, 100);  
    private Sphere projectile = new Sphere(10);  
    private final int DEFAULT_Y = HEIGHT - HEIGHT/6;  
    private int projectileY = DEFAULT_Y;  
    ...  
    private void update() {  
        if (projectileY < -20) projectileY = DEFAULT_Y }  
        group.getChildren().clear();  
        group.getChildren().add(player);  
        if (projectileY != DEFAULT_Y) {  
            projectile.setTranslateY(projectileY);  
            group.getChildren().add(projectile);  
            projectileY -= 5;  
        }  
    }  
}
```

## Spôsob 2

Objekty si vytvoríme na začiatku a potom im iba meníme súradnice a pridávame ich do detí skupiny.

```
public class Projectiles2 extends Application {  
    private Box player = new Box(100, 100, 100);  
    private Sphere ...;  
    private final ... = HEIGHT/6;  
    private int ...;  
    ...  
    private void ...  
        if (pro ... DEFAULT_Y }  
        group.g ...  
        group.g ...  
        if (pro ...  
            projectile.setTranslateY(projectileY);  
            group.getChildren().add(projectile);  
            projectileY -= 5;  
        }  
    }  
}
```




Objektom iba meníme súradnice a oni sa prekresľujú automaticky!!!

```
public class Projectiles3 extends Application {
    private Box player = new Box(100, 100, 100);
    private Sphere projectile = new Sphere(10);
    private final int DEFAULT_Y = HEIGHT - HEIGHT/6;
    ...
    private void update() {
        if (projectile.getTranslateY() < -20) {
            projectile.setTranslateY(DEFAULT_Y);
        }
        if (projectile.getTranslateY() != DEFAULT_Y) {
            projectile.setTranslateY(
                projectile.getTranslateY() - 5);
        }
    }
}
```

Objektom iba meníme súradnice a oni sa prekresľujú automaticky!!!

```
public class Projectiles3 extends Application {  
    private Box player = new Box(100, 100, 100);  
    private Sphere projectile;  
    private final double HEIGHT = 6;  
    ...  
    private void update() {  
        if (projectile != null) {  
            projectile.translate(velocity.x, velocity.y, 0);  
        }  
        if (projectile != null && projectile.getY() > HEIGHT) {  
            projectile.translate(0, 0, -1);  
        }  
    }  
}
```



- Komplikácia nastáva, keď sa môžu objekty dynamicky objavovať/zanikať.
- Vtedy potrebujeme nejako spravovať obsah množiny `group.getChildren()`.
- Určite totiž nechceme, aby nám mimo zorného pola lietali stovky projektilov, alebo ich nejako špinavo skrývať.
- Ako ale rozlišovať medzi objektami? Ktoré si máme nechať a ktoré vyhodiť? Keď si pamätáme iba súradnice v nejakých vlastných štruktúrach a v každom frame deti premažeme, môžeme si byť istý, že sme na nič nezabudli...
- Samozrejme sa to dá riešiť - napr. vytvorením vlastnej triedy na reprezentáciu projektilov podedenej od `Sphere/Circle`.

# Mazanie objektov

V každom frame vymažeme sféry, ktoré sú za vrchným okrajom obrazu.

```
private void shoot() {
    Sphere projectile = new Sphere(10);
    projectile.setTranslateX(WIDTH / 2);
    projectile.setTranslateY(DEFAULT_Y);
    group.getChildren().add(projectile);
}

private void update() {
    Set<Node> toRemove = new HashSet<>();
    for (Node n : group.getChildren()) {
        if (n instanceof Sphere) {
            Sphere s = (Sphere) n;
            if (s.getTranslateY() < -20) toRemove.add(n);
            else s.setTranslateY(s.getTranslateY() - 5);
        }
    }
    group.getChildren().removeAll(toRemove);
}
```

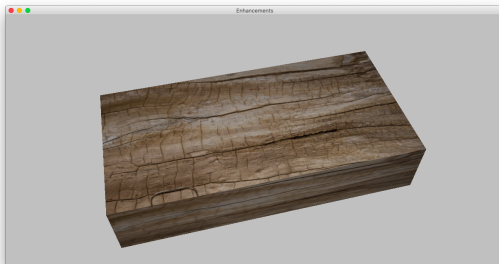
- 1 Základy (3D v JavaFX, pozícia kamery, group/pane)
- 2 Kamera (projekcie, depth buffer, generovanie prostredia)
- 3 Objekty (správa objektov, aktualizovanie pozície, mazanie)
- 4 Vylepšenia (materiály, svetlá, transformácie, UI)**
- 5 Modely a GUI (import, vlastné modely, UI panely)
- 6 Interakcia (pohyb hráča, rotácie, kolízie, raycast, ukážky)

- V 2D vieme objektom nastavovať farbu, prípadne gradient, či textúru z obrázku.
- 3D objekty ponúkajú širšie možnosti, môžete im nastaviť materiál.
- Jediná implementácia materiálu v JavaFX je `PhongMaterial`. Je teoreticky možné implementovať si vlastný materiál. Phongov reflektčný model spoločne s phongovým shading modelom sú **štandardom** v PC grafike.
- Reflektčný model určuje, akým spôsobom a z akých zložiek sa v scéne počíta osvetlenie objektov a shading model zasa hovorí o tom, ako sa rôzne intenzity svetla zobrazia na objekte.
- Pre praktické využitie je pre nás dôležité, že materiálu môžeme nastaviť farbu, textúru (diffuse map) a aj bump a specular mapu. Tie ovplyvnia správanie svetla. Bump mapa vytvorí ilúziu nerovností povrchu a specular mapa zasa ovplyvňuje odrazivosť svetla.



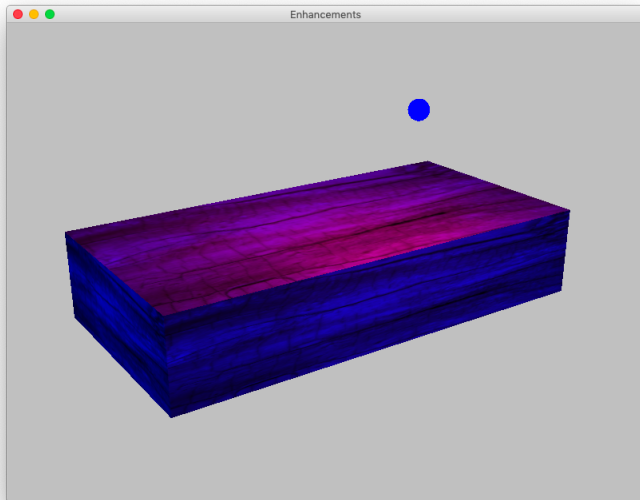
# Ukážka materiálov

```
private void createBox() {  
    PhongMaterial material = new PhongMaterial();  
    //material.setDiffuseColor(Color.ORANGE);  
    material.setDiffuseMap(new Image("file:wood.jpg"));  
    material.setBumpMap(new Image("file:bump.jpg"));  
    Box box = new Box(100, 20, 50);  
    box.setMaterial(material);  
    group.getChildren().add(box);  
}
```



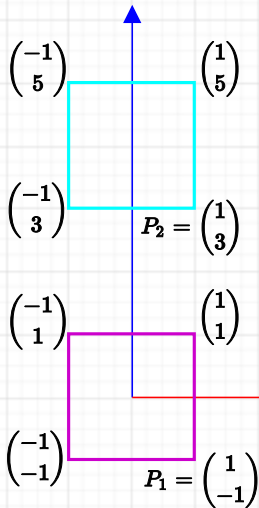
- Ďalej vieme do scény pridávať svetlá. Pozornejší si mohli uvedomiť, že JavaFX má nejaké svetlá v scéne by default (inak by bolo všetko čierne). Akonáhle do scény pridáme nejaké svetlo, default svetlá sa deaktivujú.
- Prvým typom svetla je AmbientLight. To je všadeprítomné svetlo, ktoré rovnomerne osvetľuje úplne všetko v scéne, nehl'adiac na smer. V grafike sa toto pri Phongovom reflektčnom modeli, ktorý neuvažuje odraňžanie svetla, používa ako trik na to, aby nikde nebola úplná tma. Zväčša sa preto používa v kombinácii s iným svetlom.
- PointLight má svoju lokáciu, z ktorej svieti rovnomerne do všetkých smerov.
- Obom svetlám vieme nastaviť farbu.

```
private void createLights() {  
    PointLight pointLight = new PointLight();  
    pointLight.setColor(Color.YELLOW);  
    pointLight.getTransforms().add(new Translate(0, -50, 100));  
    pointLight.setRotationAxis(Rotate.X_AXIS);  
  
    AnimationTimer timer = new AnimationTimer() {  
        @Override  
        public void handle(long now) {  
            pointLight.setRotate(pointLight.getRotate() + 1);  
        }  
    };  
    timer.start();  
  
    AmbientLight ambientLight = new AmbientLight(Color.WHITE);  
    group.getChildren().addAll(pointLight, ambientLight);  
}
```



- V predchádzajúcej ukážke sme pozíciu `pointLight` nastavili inak než s `pointLight.setTranslateY(double val);`. To je totiž zjednodušený príkaz, ktorý so zmenou pozície posunie aj ukotvenie.
- Každý objekt má zoznam transformácií, ktoré sú na neho aplikované. Príkazom `pointLight.getTransforms().add(new Translate(0, -50, 100));` sme svetlo posunuli o 50 jednotiek na y-ovej osi, no jeho ukotvenie ostalo tam kde bolo, čiže v počiatku súr. systému.
- V animácii potom zmenou rotácie príkazom `pointLight.setRotate(pointLight.getRotate() + 1);` svetlo rotuje vzhľadom na svoje ukotvenie, čiže orbituje okolo počiatku systému, v ktorom sa nachádza naša drevená doska.
- Transformácie sú v **grafike** téma sama o sebe a nie je priestor na detailné vysvetlenie. Netrápte sa tým - experimentujte, googlite, pýtajte sa...

- Pozície zapisujeme ako vektory  $\begin{pmatrix} x \\ y \end{pmatrix}$ .

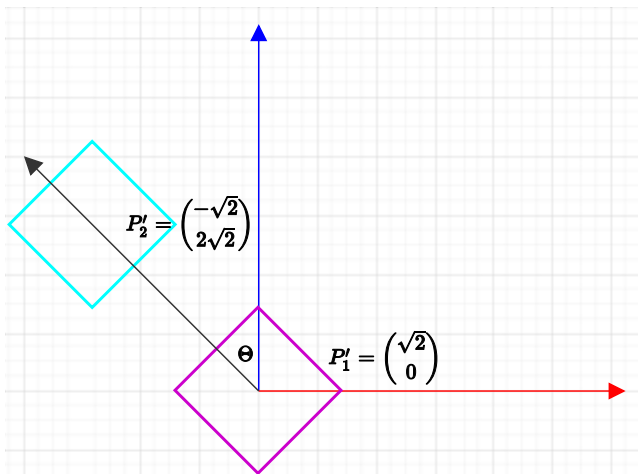


$$R = \begin{pmatrix} \cos\Theta & -\sin\Theta \\ \sin\Theta & \cos\Theta \end{pmatrix}, \Theta = 45^\circ \quad R = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$$

$$P'_1 = R \cdot P_1 = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{2}} \\ 0 \end{pmatrix} = \begin{pmatrix} \sqrt{2} \\ 0 \end{pmatrix}$$

$$P'_2 = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 3 \end{pmatrix} = \begin{pmatrix} -\sqrt{2} \\ 2\sqrt{2} \end{pmatrix}$$

- Vidno abstrakciu ukotvenia objektu (globálny stred sústavy).
- Toto matematické okienko berte len ako ukážku pre lepšiu predstavu!



- 1 Základy (3D v JavaFX, pozícia kamery, group/pane)
- 2 Kamera (projekcie, depth buffer, generovanie prostredia)
- 3 Objekty (správa objektov, aktualizovanie pozície, mazanie)
- 4 Vylepšenia (materiály, svetlá, transformácie, UI)
- 5 Modely a GUI (import, vlastné modely, UI panely)**
- 6 Interakcia (pohyb hráča, rotácie, kolízie, raycast, ukážky)

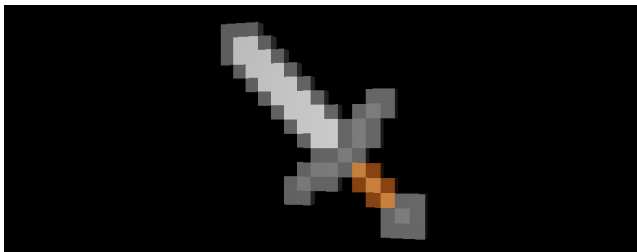


- JavaFX priamo nepodporuje import externých 3D modelov, existuje na to však samostatné **JavaFX 3D model importers knižnice**.
- .jar knižnicu v IntelliJ pridáte kliknutím na File → Project Structure... → Libraries → New Project Library (ikona +).
- 3D objekt a textúry umiestnite do koreňovej zložky projektu.

```
TdsModelImporter tdsImporter = new TdsModelImporter();  
tdsImporter.read("hst.3ds");  
Node[] rootNodes = tdsImporter.getImport();  
group.getChildren().addAll(rootNodes);
```

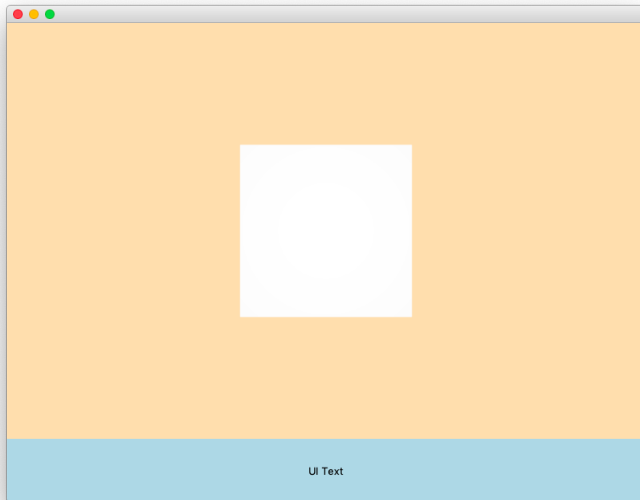
# Vlastný model

- Druhý spôsob je poskladať si vlastný model z primitív (kociek).
- Optimálne je zaobaliť to do samostatnej triedy, ktorá dedí od Group, takže viete s vaším modelom pracovať ako so samostatným objektom.
- Vymyslite si nejaký rozumný spôsob reprezentácie, ukážkový kód používa napr. vykreslenie kociek do  $16 \times 16$  mriežky.
- Model si viete aj nejako zapísať do súboru, podobne ako mapu.



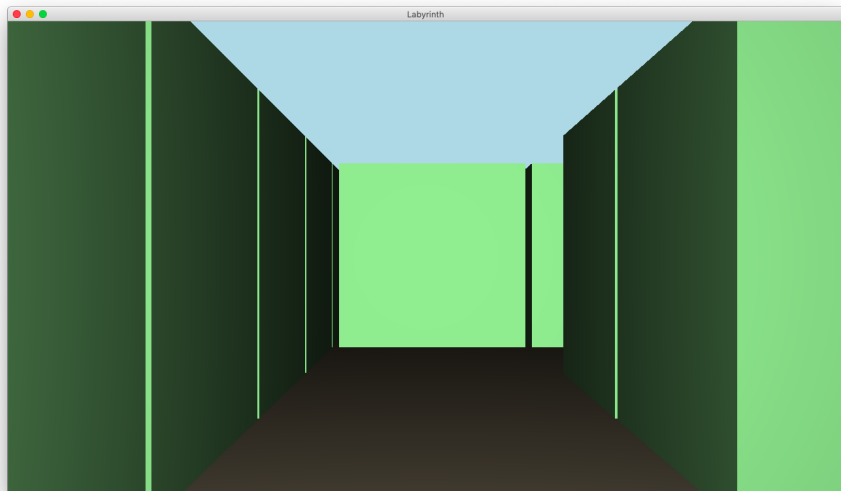
- 3D hry mávajú 2D užívateľské rozhranie.
- Môžeme si vytvoriť HUD - heads up display. Vytvoríme Group z 2D objektov (napr. Text) a jeho pozíciu nabindujeme na kameru tak, aby hráč videl informácie vždy pred sebou. (čo keď dokráčame k stene a UI zmizne v nej? Problém...)
- Asi je jednoduchšie vytvoriť UI ako samostatný **panel** niekde naboku.
- Na to využijeme BorderPane, do stredu dáme SubScene 3D prostredia.

```
public class UserInterface2D extends Application {  
    // 2D UI Panel  
    private HBox panel = new HBox();  
    Group group = new Group(new Box(20,20,20));  
    // 3D Scene  
    private SubScene scene =  
        new SubScene(group, ..., SceneAntialiasing.BALANCED);  
    private BorderPane layout = new BorderPane();  
    private Scene root = new Scene(layout, 1280, 720);  
    @Override  
    public void start(Stage primaryStage) {  
        preparePanel();  
        layout.setCenter(scene); // hore bude 3D scena  
        layout.setBottom(panel); // dole 2D UI Panel  
        ...  
    }  
    ...  
}
```



- 1 Základy (3D v JavaFX, pozícia kamery, group/pane)
- 2 Kamera (projekcie, depth buffer, generovanie prostredia)
- 3 Objekty (správa objektov, aktualizovanie pozície, mazanie)
- 4 Vylepšenia (materiály, svetlá, transformácie, UI)
- 5 Modely a GUI (import, vlastné modely, UI panely)
- 6 Interakcia (pohyb hráča, rotácie, kolízie, raycast, ukážky)

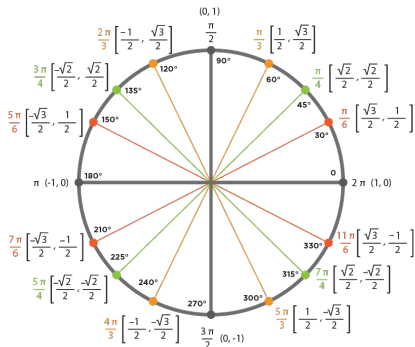
# Labyrinth



- Spojme nadobudnuté poznatky a vytvorme aplikáciu, v ktorej sa hráč pohybuje po labyrinte.
- Vykreslenie prostredia sme už videli v ukážke rozdielu medzi perspektívnym a paralelným zobrazením. Pohyb hráča vyriešime jednoducho zmenou súradníc kamery pri stlačení pohybových kláves.
- Čo s **otáčaním** hráča? Ak iba otočíme kameru vzhľadom na os Y, pohyb nebude sedieť. Hráč sa bude pozeráť doprava, no pohyb dopredu bude stále v rovnakom smere. Je to akoby ste vedeli otáčať iba hlavou, no telo by ostalo natočené stále rovnako.
- V 2D tento problém nenastáva, lebo sa hráč väčšinou pohybuje len v smere osí systému. V 3D sa potrebujeme hýbať s v smere, v akom sme natočení... Ako vypočítať zmenu súradníc?



**Možnou** odpoveďou sú goniometrické funkcie a jednotková kružnica. Pamätáme si uhol natočenia, začíname na nule. Posun po ose X určuje sínus tohto uhla a po Y kosínus. Týmto sa pohneme vždy o jednotku vzdialenosti, ak chceme o viac, posuny pre násobíme. Ak by sme sa chceli pozerat' aj hore/dole a hýbať sa v tomto smere (lietadlo), posuny by sme počítali z dvoch kružníc...



# Ukážka pohybu hráča

Netreba zabudnúť na to, že zabudované funkcie `Math.sin/Math.cos` berú uhol v radiánoch...

```
private void cameraMovement(KeyEvent keyEvent) {  
    double dx = Math.sin(  
        Math.toRadians(camera.getRotate()));  
    double dz = Math.cos(  
        Math.toRadians(camera.getRotate()));  
  
    switch (keyEvent.getCode()) {  
        case W:  
            camera.translateZProperty().set(  
                camera.translateZ() + dz);  
            camera.translateXProperty().set(  
                camera.translateX() + dx);  
            break;  
    }  
}
```

Otáčať sa môžeme klávesami, ale napríklad aj pomocou myši. To je v hrách štandardný prístup.

```
scene.setOnMousePressed(event -> {  
    mousePressed = true;  
    lastMouseX = event.getSceneX();  
});
```

```
scene.setOnMouseReleased(event -> mousePressed = false);
```

```
scene.setOnMouseDragged(event -> {  
    if (mousePressed) {  
        camera.setRotationAxis(Rotate.Y_AXIS);  
        camera.setRotate(camera.getRotate() +  
            (event.getSceneX() - lastMouseX)  
            * mouseSensitivity);  
    }  
    lastMouseX = event.getSceneX();  
});
```

- Zatiaľ môžeme v pohode nakráčať do steny labyrintu a prejsť skrz.
- Kolízie môžeme riešiť štandardne porovnávaním súradníc. Vieme získať stred objektu a z toho vypočítať okrajové body. V prípade kocky napríklad od stredu odpočítame/pripočítame polovicu dĺžky strany (Corners.java). Je to analógia k 2D, kde máme ľavý horný a pravý dolný bod štvorca. Porovnaním so súradnicami kamery vieme zistiť, či kamera kolидуje s objektom.
- V JavaFX sa to ale dá aj jednoduchšie pomocou zabudovanej metódy `intersects(...)`. Nasledujúca ukážka sa dá rovnako využiť aj v 2D.
- Ale **pozor**, Bounds sú vždy zarovnané s osami systému, to znamená, že nefungujú ako by ste čakali pri zrotovaných objektoch (pozri BoundsTest.java)!

# Ukážka výpočtu kolízií

```
private void cameraMovement(KeyEvent keyEvent) {
    double oldZ = camera.translateZ();
    double oldX = camera.translateX();
    switch (keyEvent.getCode()) {...}
    if (checkCollisions()) {
        camera.translateZProperty().set(oldZ);
        camera.translateXProperty().set(oldX);
    }
}

private boolean checkCollisions() {
    for (Node n: group.getChildren()) {
        if (n instanceof Box) {
            Box b = (Box) n;
            if (b.getBoundsInParent().
                intersects(camera.getBoundsInParent()))
                return true;
        }
    }
    return false;
}
```

- Obcaš chceme s objektami v scéne interagovať pomocou myši.
- V 2D je to v podstate jednoduché, keďže máme 2D pozíciu myši na obrazovke a 2D scénu.
- V 3D to nie je také priamočiare a na tento účel sa používa raycasting.
- Vieme si to predstaviť ako vyslanie lúča od stredu kamery, na smer určený kliknutím a výpočet kolízie tohto lúča s nejakým objektom.
- Našťastie to nemusíme implementovať (kolízia priamky a roviny), JavaFX to už má implementované, súradnice získame z MouseEvent zavolaním `getX()`, `getY()`, `getZ()`.

```
scene.setOnMousePressed((MouseEvent me) -> {  
    Sphere s = new Sphere(2);  
    s.setTranslateX(me.getX());  
    ...  
    s.setMaterial(sphereMaterial);  
    group.getChildren().add(s);  
});
```

- Do labyrintu viete jednoducho dorobiť strielanie projektílov z pozície hráča a máte už skoro kompletnú strielačku. Môžete sa pritom inšpirovať odbočkou o dynamických objektoch :).
- S 3D sa v JavaFX pracuje v podstate rovnako ako s 2D. Akurát si treba **zvyknúť** na tretiu os.
- Netreba sa toho báť, výsledky sú pôsobivé. Hlavne si všetko kreslite a nenechajte sa popliesť smermi...
- Ďalšie dobré ukážky nájdete na:  
<https://github.com/afsalashyana/JavaFX-3D> respektíve  
<https://www.genuinecoder.com/category/javafx/javafx-3d/>.

- Začiatkom budúceho roka si budete vyberať **bakalársku prácu**, pričom prehľad o grafike vám môže pomôcť s výberom.
- Každoročne sú na výber zaujímavé témy od firmy **Skeletex** (skeletex.xyz). Urobte si prehľad už teraz a cez leto môžete rozmýšľať.
- Okruhy tém: spracovanie 3D dát, vytváranie kostier, snímanie pohybu, GPU shadery, strojové učenie...
- Šanca naučiť sa používať technológie ako Unreal Engine 4, CUDA, OpenGL... Väčšinou v C++.



# Bakalárka II

