

Základy tvorby interaktívnych aplikácií

Architektúra, MVC, HTML Canvas

Ing. Peter Kapec, PhD.

LS 2019-20

Obsah

- Architektúra interaktívnych web-aplikácií
 - návrhový vzor MVC
- HTML Canvas
 - kreslenie geom. tvarov a obrazov
- Ukážky implementácie v JavaScript
 - základná varianta
 - rozšírenie o viacero objektov a animáciu
 - štruktúra pre komplexnejšiu aplikáciu

Architektúra int. web-aplikácií

Architektúra int. web-aplikácií

- Potreba určenia základnej štruktúry aplikácie
- Nie je to len rozdelenie kódu aplikácie
- Celková organizácia projektu
- Rôzne odporúčania a návrhové vzory
- SW a HW obmedzenia

Architektúra int. web-aplikácií

Logika

Grafika

Vstupy

Zvuk

Hudba

Multiplayer

Zdroje

Architektúra int. web-aplikácií

Logika

Grafika

Vstupy

Zvuk

Hudba

Multiplayer

Zdroje

- Javascript <script>
- Kreslenie <canvas>
- onkeydown ...
- <audio>
- <audio>
- WebSocket
- Rieši browser

Počítačová grafika a HTML Canvas

2D Grafika

- Rastrová a vektorová reprezentácia
- Založené na vykresľovaní
 - základných geom. útvarov
 - rastrových obrázkov
- farebný model RGBA
 - kompozícia na základe priesvitnosti

2D Grafika



Street
Fighter



Earthworm
Jim

3D Grafika

- Založená na vykresľovaní povrchovej reprezentácie (najčastejšie polygóny a trojuholníky)
- Objekty sú osvetlené a otextúrované
- Oveľa výkonovo náročnejšia
- Viac na predmete
Princípy počítačovej grafiky a spracovania obrazu
- V HTML5 možno použiť *Canvas* a *WebGL*
 - knižnica *three.js*

HTML Canvas

- Štandardný HTML5 element, ktorý reprezentuje voľne manipulovateľný obraz (2D frame buffer)
- Poskytuje kontext, cez ktorý možno kresliť do plochy priamo z JavaScriptu
- *Context* poskytuje základné funkcie pre kreslenie
- **Nie je nutné manipulovať DOM**
- Rýchlejšie a vhodnejšie pre tvorbu hier

HTML Canvas

- Poskytuje:
 - kreslenie textu
 - kreslenie geometrických telies
 - kreslenie obrázkov
- Umožňuje:
 - animácie
 - interakciu

Canvas element v HTML

- vytvorenie *Canvas*-u

```
<canvas id="canvas" height="100" width="100">  
Sorry no Canvas :(  
</canvas>
```

- práca s *Canvas*-som

```
var canvas = document.getElementById("canvas")  
var context = canvas.getContext("2d")
```

Vytvorenie kontextu

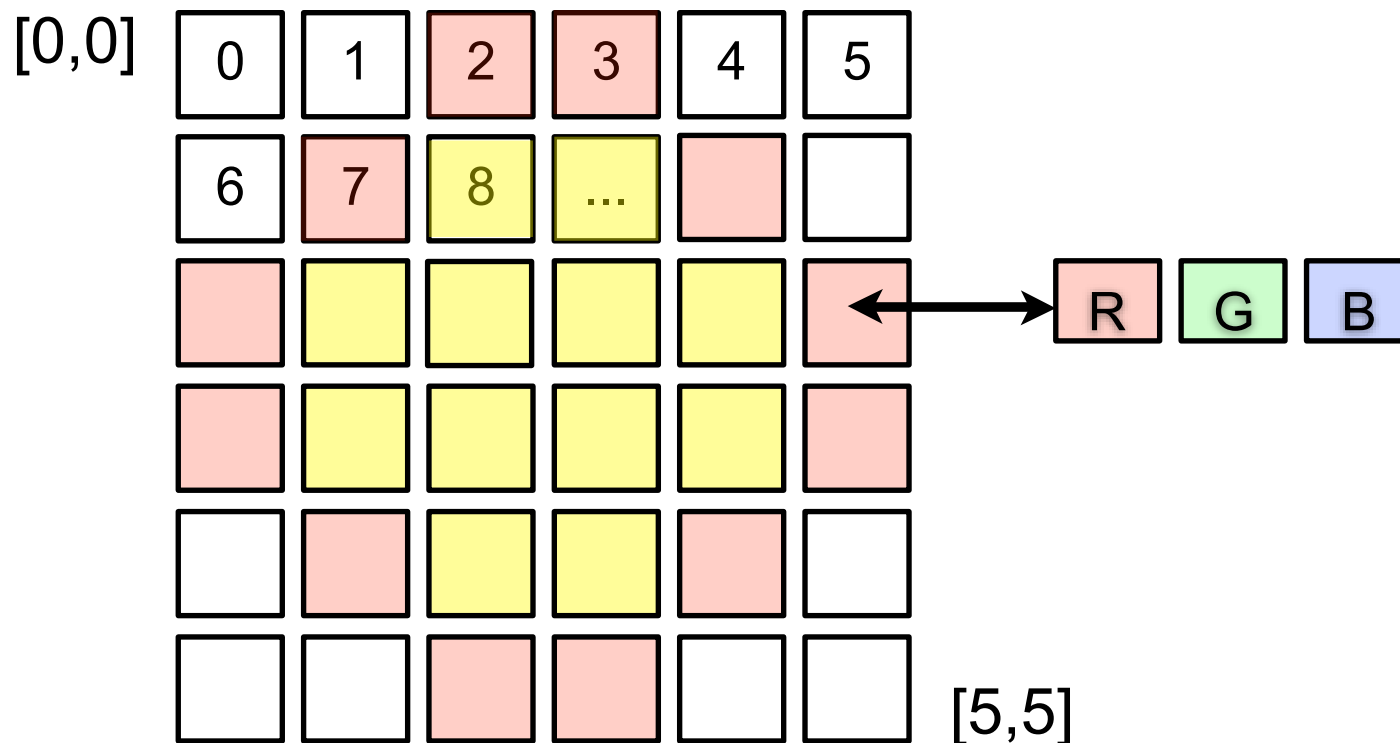
- Na prácu so samotným *Canvas* potrebuje získať jeho *Context* objekt

```
var canvas = document.getElementById("canvas")  
var context = canvas.getContext("2d")
```

- Context objekt poskytuje
 - atribúty - nastavenie farby, typ čiary, výplne, ...
 - metódy - vykreslenie čiary, kruhu, krivky, obrázka,...
- Obráz sa nemení pokiaľ ho neupravíme

Canvas, framebuffer, pixely

- Canvas poskytuje framebuffer - 2D pole pixelov
 - ľavý horný roh $[0,0]$, pravý dolný roh $[\text{height}, \text{width}]$
- Každý pixel možno priamo meniť
 - súradnica $[x,y]$, alebo index v rozsahu 0 až *img.data.length*



Prístup k pixelom

- Možno priamo pracovať s pixelom
 - zmena farby prvého pixela

```
// nacistanie obrazu 100x100 pixelov zo suradnice [0,0]
var img = canvas.getImageData(0,0,100,100)
img.data[0] = 255; // R
img.data[1] = 0; // G
img.data[2] = 0; // B
img.data[3] = 0; // A
canvas.putImageData(img, 0, 0) // zapisanie obrazu 100x100
pixelov na
                                // na suradnicu [0,0]
```


Kreslenie textu

- Vykreslenie textu do stredu plochy

```
var x = canvas.width / 2;  
var y = canvas.height / 2;  
context.font = '30pt Calibri';  
context.textAlign = 'center';  
context.fillStyle = 'blue';  
context.fillText('Hello World!', x, y);
```



Hello World!

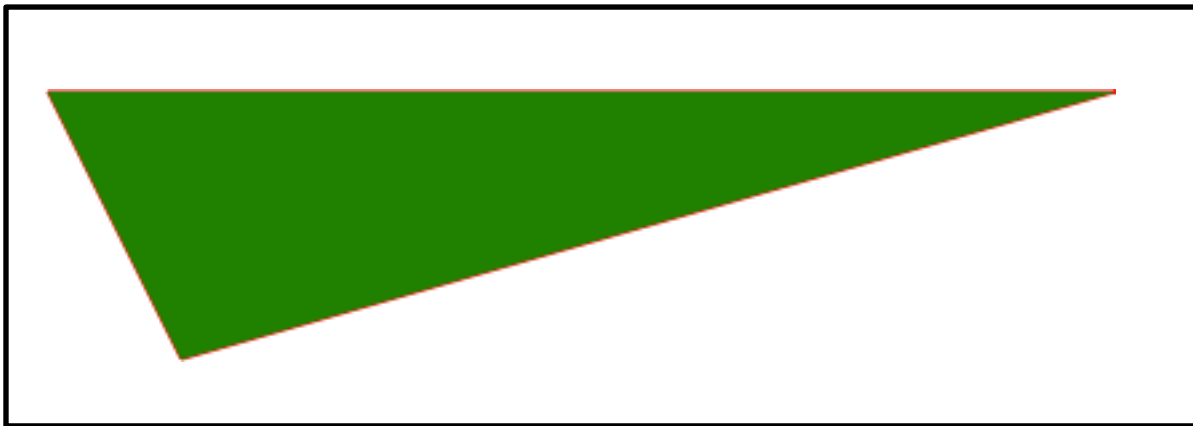
Kreslenie čiar a geom. tvarov

- Canvas podporuje kreslenie čiar a tvarov

```
context.beginPath();           // začni kreslit tvar
context.strokeStyle = "red";   // nastav farbu ciary
context.fillStyle = "green";   // nastav farbu vyplne
context.moveTo(100, 150);      // presun sa na poziciu
context.lineTo(450, 50);       // nakresli ciaru na poziciu
context.lineTo(50, 50);
context.lineTo(100, 150);
context.closePath();           // spoj posledny bod s prvym
context.stroke();              // vykresli
```

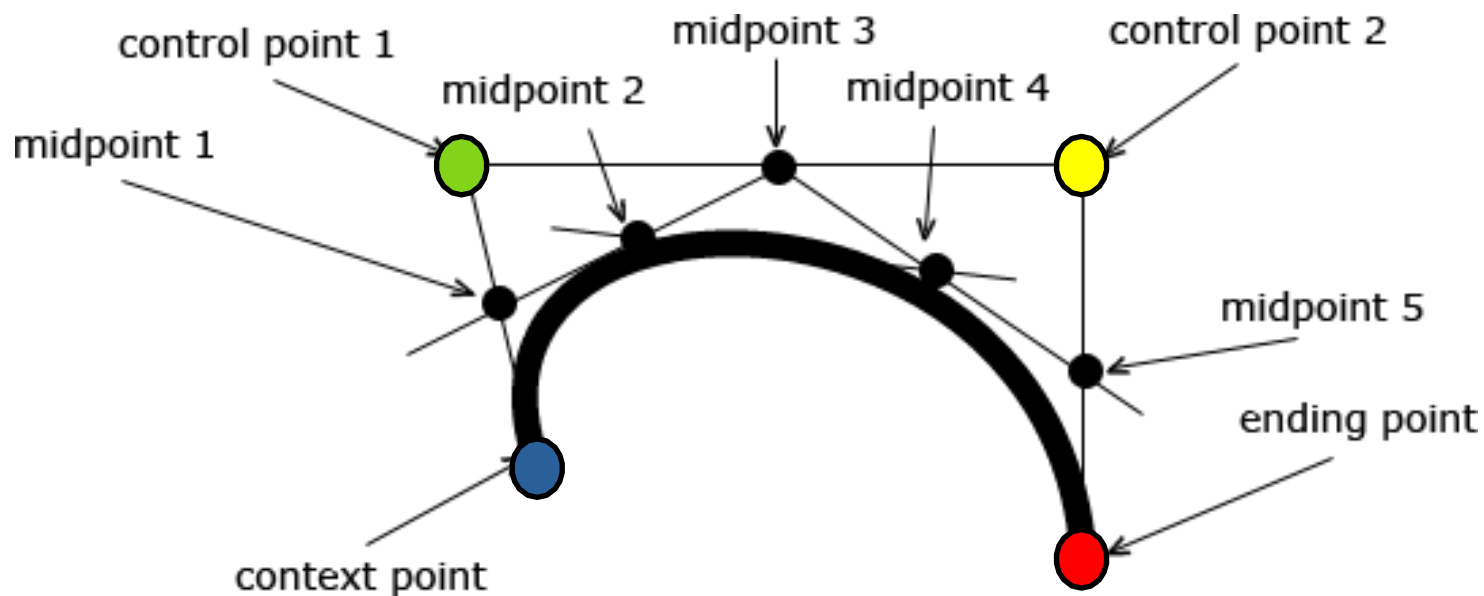
Kreslenie čiar a geom. tvarov

- Canvas podporuje kreslenie čiar a tvarov



Kreslenie kriviek

- Možno kresliť Bezierove krivky



```
context.beginPath();  
context.moveTo(188, 130);  
context.bezierCurveTo(140, 10, 388, 10, 388, 170);  
context.lineWidth = 10;  
context.strokeStyle = "black";  
context.stroke();
```

Obrázky

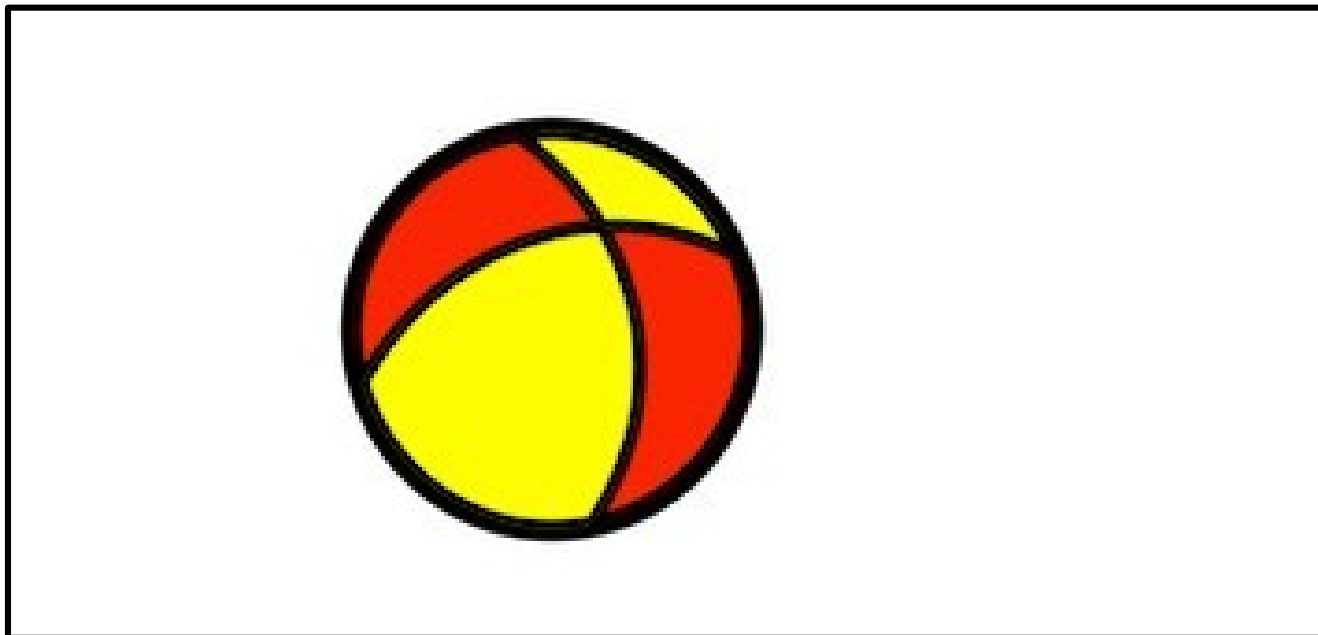
- Obrázky je možné kresliť pomocou objektu *Image*
- Je možné použiť `` element z DOM

```
var imageObj = new Image();
imageObj.src = "ball.png"

// Obrázok sa vykresli až po nacistani
imageObj.onload = function () {
    context.drawImage(imageObj, x, y, width, height);
}
```

Obrázky

- Obrázky je možné kresliť pomocou objektu *Image*
- Je možné použiť `` element z DOM



Manažment stavov

- *Context* dokáže uchovať a obnoviť svoj stav

- užitočné pri kreslení viacerých nezávislých objektov
- stav sa ukladá-vyberá zo zásobníka

```
// vykresli objekt1
context.save()                // uloz akt. stav
// vykonaj transformacie nad objekt1
context.drawImage(imageObj1,0,0) // vykresli transformovany
obr.
context.restore()             // obnov stav

// vykresli objekt2
context.save()                // uloz akt. stav
// vykonaj transformacie nad objekt2
context.drawImage(imageObj2,0,0) // vykresli transformovany
obr.
context.restore()             // obnov stav
```

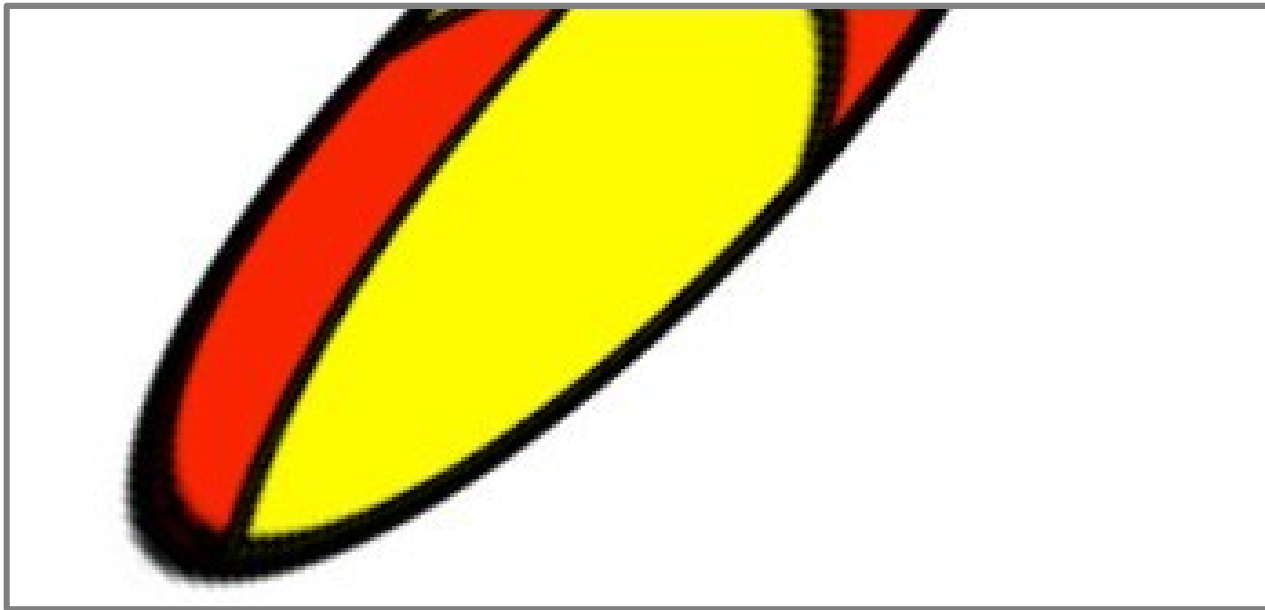
Transformácie

- Je možné aplikovať základné geometrické transformácie

```
// kresli objekt
context.save()                // ulozenie akt. stavu
context.translate(40,120)     // posun
context.rotate(-45*Math.PI/180.0) // rotacia
context.scale(3.0,1.0)        // skalovanie
context.drawImage(imageObj,0,0) // vykreslenie
transformovaneho obr.
context.restore()             // obnovenie stavu
```


Transformácie

- Je možné aplikovať základné geometrické transformácie



Ďalšie zdroje ku HTML Canvas

- <http://www.html5canvastutorials.com/>
- https://www.w3schools.com/graphics/canvas_reference.asp
 - podrobný opis všetkých atribútov a metód

Základné princípy

- Každý obraz vytvárame celý z prázdneho obrazu
- Každý obraz kreslíme od pozadia k poprediu ako kompozíciu viacerých častí
- Ak vykresľujeme aspoň 24 obrázkov za sekundu tak vytvárame ilúziu plynulej animácie
- Animáciu vytvárame zmenou parametrov **Modelu** a jeho následným vykreslením cez **View**

Kompozícia obrazu

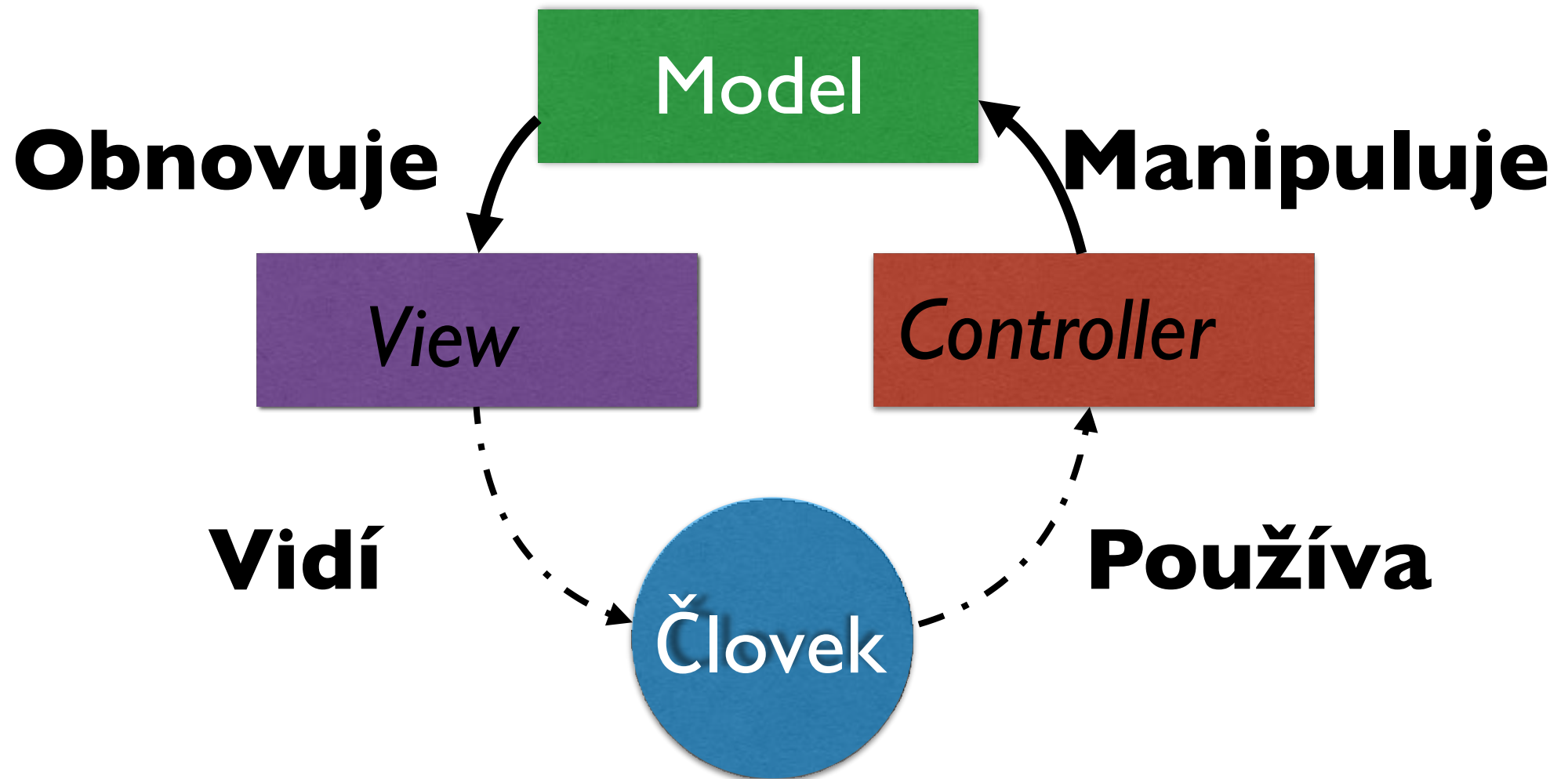


Návrhový vzor MVC

Vzor Model-View-Controller

- Základným problémom je určiť aká časť kódu je zodpovedná za aké operácie
- Návrhové vzory popisujú vysoko-úrovňovú organizáciu, ktorá rieši bežné problémy
- V objektovo orientovanom programovaní je návrhový vzor základom, z ktorého sa vychádza

Vzor MVC



Vzor MVC

- **MVC – Model, View, Controller**
- **Model** – vnútorná reprezentácia dát aplikácie
- **View** - pohľad na model a jeho zobrazenie
- **Controller** – riadenie a spracovanie vstupov a zmien
- Zvyčajne sa tieto časti implementujú pomocou oddelených objektov alebo modulov

Model

- Väčšina programov má v prvom rade vykonávať prácu a nemusí dobre vyzerat'
 - Existujú samozrejme výnimky
 - Užitočné aplikácie existovali dávno pred GUI
- Model je časťou, ktorá vykonáva riešenie, všetku prácu – je modelom riešenia problému

Model

- Model by mal byť nezávislý od ostatných komponentov
 - Poskytuje však rozhranie (metódy, funkcie), ktoré možno použiť

Controller

- **Controller** určuje ako sa bude s **Modelom** pracovať
 - Častokrát je **Controller** samotné GUI
- Je skoro vždy možné **Controller** a **Model** oddeliť
- Návrh **Controller-a** je však zvyčajne závislý od **Modelu**
- **Model** by sa nemal nikdy prispôsobovať **Controller-u**

View

- Používateľ očakáva, že bude vidieť stav aplikácie
- **View** poskytuje náhľad na to čo **Model** vykonáva
 - **View** je pasívny a neovplyvňuje **Model**
- **Model** je zväčša nezávislý od **View**, avšak poskytuje mu rozhranie (funkcie/metódy)
- **Pohľad** by *nemal zobrazovať* nič súvisiace s činnosťou **Controller**

View a Controller

- Častokrát je však užitočné spojiť **Controller** a **View**, hlavne u malých aplikácií
- Je to vhodné napríklad pri vysokom prepojení **Controller** s **View**
- **Model** však vždy zostáva oddelený
- **Nikdy** nemiešajte GUI kód s **Modelom**

Oddelenie závislostí

Oddelenie závislostí

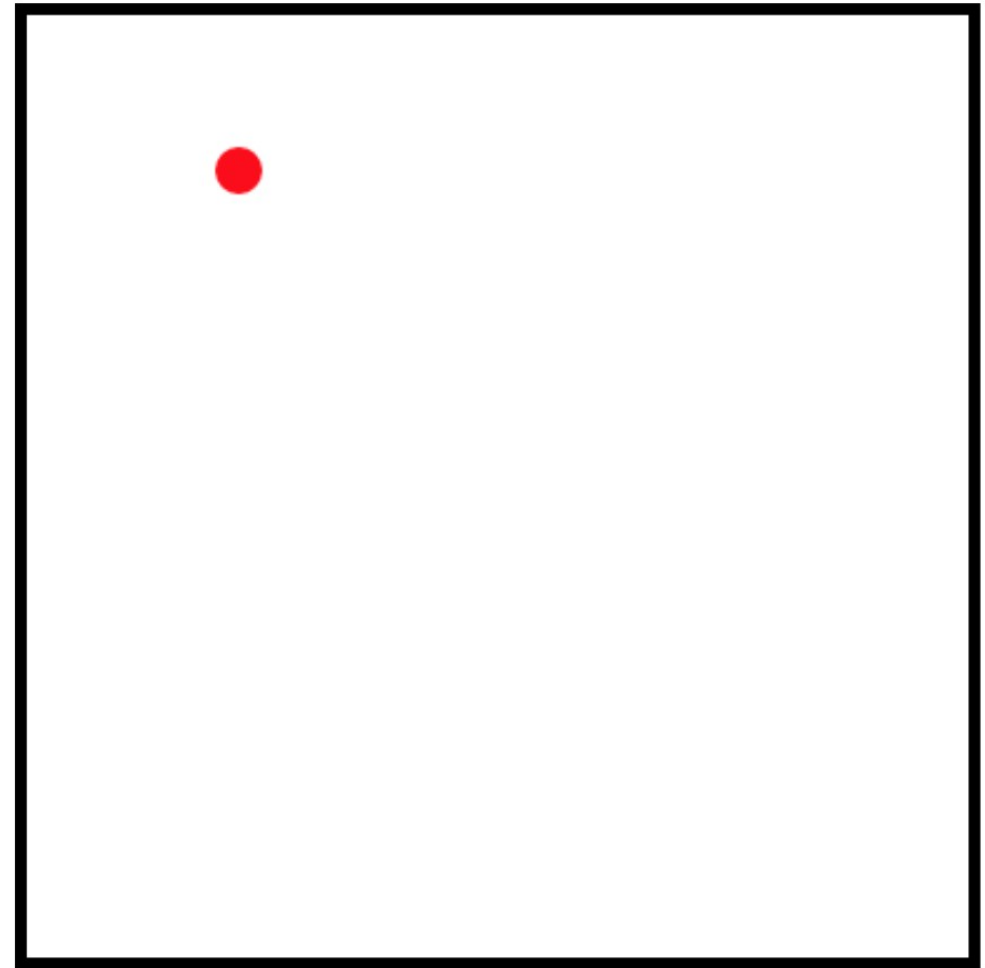
- Nezávislosť častí kódu vedie k robustnosti, flexibilitě pri zmenách a ľahšej udržiavateľnosti kódu
- Nezávislosť častí kódu je žiadúca vlastnosť
- Z predmetu *Princípy softvérového inžinierstva*:
 - Súdržnosť - ako silno je softvérový prvok “fokusovaný”
 - Previazanosť - ako silne je softvérový prvok spojený s inými prvkami

Oddelenie závislostí

- **Model** by nemal byť kontaminovaný ovládaním či kódom vykreslovania
- **View** by mal reprezentovať **Model** taký aký naozaj je bez skreslenia
- **Controller** by mal komunikovať s **Modelom** a **View** len za účelom ich manipulácie
- **Controller** môže napríklad nastaviť premenné, ktoré **View** a **Model** používa

Dot

- Ukážková aplikácia, demonštrácia MVC
- Stlačenie tlačidla *krok* vykoná posun bodu
- Bod sa odráža od hraníc obrazu
- Krok a pozícia sú zobrazené v texte
- Situácia je ilustrovaná obrazom



Step

Tick: 4

Dot

- **Model** ovláda pohyb bodu
- V tomto prípade musí **Model** vedieť veľkosť **View**
 - Pretože potrebuje modelovať hranice od ktorých nastane odraz
- **Model** nevie nič o zvyšku aplikácie a jej ovládacích prvkoch
- **Controller** spracúva stlačenie tlačítka a aktualizuje **Model**
- **View** zobrazuje scénu a počet krokov

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bouncing Dot</title>
    <script>

    </script>
    <style>
      #canvas {
        border-style: solid;
        border-width: 5px;
      }
    </style>
  </head>

  <body>
    <p>
      <canvas id="canvas" height="400" width="400"></canvas>
    </p>
    <p>
      <button id="button">Step</button>
    </p>
    <p id="text">
      Please click Step!
    </p>
  </body>
</html>
```

Celá aplikácia

```
var canvas
var ctx
var tick = 0 // pocitanie krokov po stlaceni tlacitka

// MODEL
//

// View
//

// Controller
//

// Initialize the application
window.onload = function() { // Main
    // Set up global variables for easy access
    button = document.getElementById("button") // objekt pre tlacitko
    text = document.getElementById("text") // objekt pre text
    canvas = document.getElementById("canvas") // objekt pre canvas
    ctx = canvas.getContext("2d") // objekt pre contex
    button.onclick = step // stlacenie tlacitka zavola step()
}
```

Model

```
// Simple object that stores where the dot is and what it can do
var dot = {
  x: 50, // pozicia x,y
  y: 50,
  dx: 10, // posun v smere dx,dy
  dy: 4,

  // A method to move the object
  move: function() {
    if (this.x >= canvas.width || this.x <= 0) {
      this.dx *= -1
    }
    if (this.y >= canvas.height || this.y <= 0) {
      this.dy *= -1
    }

    // Update object position
    this.x = this.x + this.dx
    this.y = this.y + this.dy
  }
}
```

View

- *drawDot()* vykreslí do *Canvas-u* červený bod

```
// Just takes care of drawing based on model
function drawDot() {
  // Clear the canvas
  ctx.clearRect(0, 0, canvas.width, canvas.height)

  // Render a dot
  ctx.fillStyle = "red"
  ctx.beginPath()
  ctx.arc(dot.x, dot.y, 10, 0, Math.PI * 2)
  ctx.closePath()
  ctx.fill()
}
```

View

- *setText()* modifikuje DOM a nastaví počet kroků

```
// Update html text
function setText() {
  text.innerHTML = "Tick: " + tick
}
```

Controller

- *step()* vykonáva hlavnú funkcionálnosť

```
// Controls what is being done in the application
// In this case it just orchestrates the model to move and view
// to
// render in a loop
function step() {
  tick++
  dot.move()
  drawDot()
  setText()
}
```


Dot

- Pozri implementáciu v `1_dot.html`

Architektúra int. web-aplikácií

Logika

Grafika

Vstupy

Zvuk

Hudba

Multiplayer

Zdroje

- Javascript <script>
- Kreslenie <canvas>
- onkeydown ...
- <audio>
- <audio>
- WebSocket
- Rieši browser

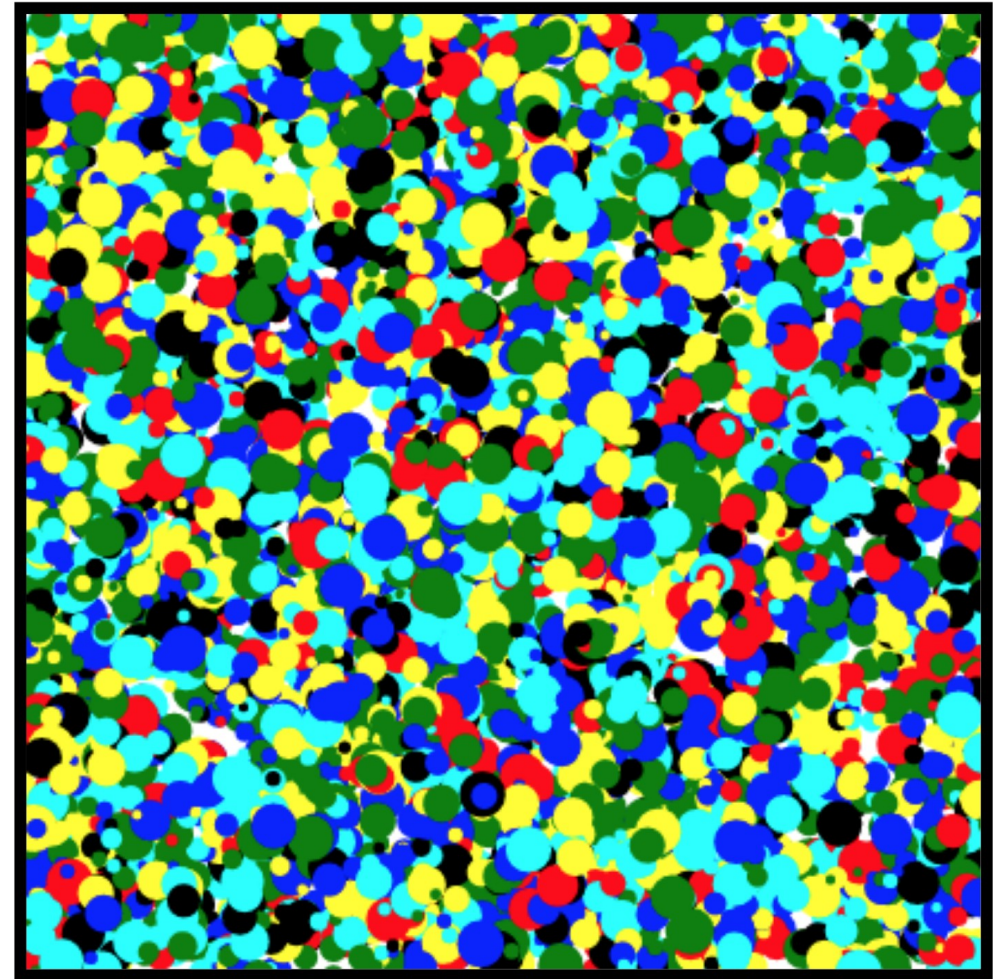
Hlavný cyklus

- Typický hlavný cyklus v hre vyzerá nasledovne JavaScript

```
function mainLoop()  
{  
    MoveScene();  
    DrawScene();  
    PlaySounds();  
    HandleInputs();  
}  
setInterval(mainLoop, 1000/60); // 60 fps
```

Rozšírenie aplikácie Dot

- Zobrazenie viacerých objektov
- Kolekcia objektov, virtuálna scéna
- Každý objekt sa inicializuje na náhodnej pozícii
 - generovanie cez:
Konštruktor
- Objekty sa pohybujú
 - animácia 30fps



Start

Rozšírenie aplikácie Dot

- Globálne premenné
 - časovač *timer*, počet krokov *tick*
 - pole farieb *colours*
 - pole objektov *dots*

```
var canvas
var ctx
var timer
var tick = 0
var colours = ["red", "green", "blue", "yellow", "cyan",
               "black"]

// Model
//
// In this case the model will be a collection of objects
var dots = []
```

Inicializácia

- tlačítko bude vykonávať funkciu *start()*
- v cykle vytvoríme inštancie *Dot* a vložíme do poľa

```
// Initialization
window.onload = function () {
    // Setup global variables
    button = document.getElementById("button")
    text = document.getElementById("text")
    canvas = document.getElementById("canvas")
    ctx = canvas.getContext("2d")
    button.onclick = start

    // Create 5000 dots
    for (i = 0; i < 5000; i++) {
        dots.push( Dot() )
    }
}
```

Model

- Objektu *Dot* v konštruktore nastavíme náhodnú pozíciu, smer pohybu a farbu

```
// This function will create a new object for each dot
function Dot() {
  // We will make a new object newDot
  var newDot = {}
  newDot.x = Math.random() * canvas.width
  newDot.y = Math.random() * canvas.height
  newDot.dx = Math.random() * 10 - 5
  newDot.dy = Math.random() * 10 - 5
  newDot.size = Math.random() * 8 + 2

  // Randomly select a colour
  var colour_index = Math.round(Math.random() * (colours.length
- 1))
  newDot.colour = colours[colour_index]

  // continue on next slide
```

Model

- Metóda *move()*:
 - otestuje či pozícia je mimo *Canvas*-u, ak je, zmení smer pohybu
 - aktualizuje pozíciu

```
// A method to move the object
newDot.move = function () {
  // Logic
  if (this.x >= canvas.width || this.x <= 0) {
    this.dx *= -1
  }
  if (this.y >= canvas.height || this.y <= 0) {
    this.dy *= -1
  }

  // Update position
  this.x = this.x + this.dx
  this.y = this.y + this.dy
}
return newDot
}
```


View

- *draw()* vykreslí všetky *Dot* objekty v poli *dots*

```
// View
//
function draw() {
    // Clear canvas
    ctx.clearRect(0, 0, canvas.width, canvas.height)

    // Render each dot
    for (i in dots) {
        var dot = dots[i]
        ctx.fillStyle = dot.colour
        ctx.beginPath();
        ctx.arc(dot.x, dot.y, dot.size, 0, Math.PI * 2);
        ctx.closePath();
        ctx.fill();
    }
}
```

Controller

- funkcia *move()* zavolá nad každým objektom *Dots* metódu *move()*
- *step()* vykoná jeden krok simulácie (animácie)

```
// Move all dots
function move() {
    for (var i in dots) {
        dots[i].move()
    }
}

function step() {
    tick++
    move()
    draw()
    setText()
}
```

Controller

- *start()*
 - prepína medzi 2 stavmi tlačítka
 - maže alebo nastavuje časovač, ktorý potom v pravidelných intervaloch volá *step()*

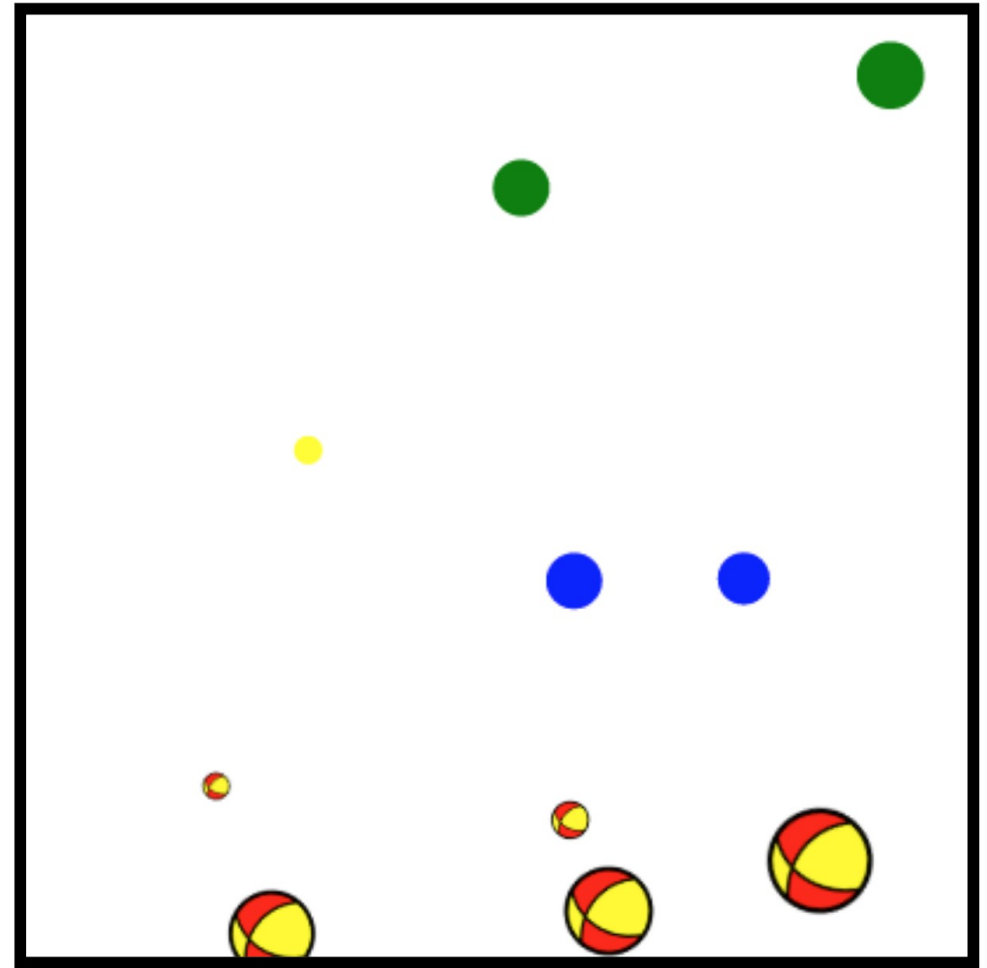
```
// Starts and stops a timer that will call step() 30 times per second
function start() {
    if (timer) {
        timer = clearInterval(timer)
        button.textContent = "Start"
    } else {
        timer = setInterval(step, 1000 / 30);
        button.textContent = "Stop"
    }
}
```

Dots

- Pozri implementáciu v `2_dots.htm`

Odrážajúca sa Lopta

- Rozdelenie kódu do súborov
 - vhodná štruktúra
 - inicializácia
 - hlavná slučka
- Viacero druhov objektov
 - nezávislé zobrazenie
 - pohyb
- Práca s časom



Základná štruktúra

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bouncing Balls</title>
    <meta charset="utf-8" />
    <link href="css/balls.css" rel="stylesheet" />
    <script src="js/ball.js"></script>
    <script src="js/dot.js"></script>
    <script src="js/main.js"></script>
  </head>
  <body>
    <p>
      <canvas id="canvas" height="400" width="400">No Canvas :(
    </canvas>
    </p>
    
  </body>
</html>
```

Globálne premenné a inicializácia

```
var canvas
var ctx
var time

// Model
var scene = []

// View / Controller / Main loop

// Initialization
window.onload = function() {
  canvas = document.getElementById("canvas")
  ctx = canvas.getContext("2d")

  for (i = 0; i < 5; i++) {           // Create 5 dots and store them in scene
    scene.push( new Dot() )
  }

  for (i = 0; i < 5; i++) {           // Create 5 balls and store them in scene
    scene.push( new Ball() )
  }
  time = Date.now()
  requestAnimationFrame(step)
}
```

Main loop

- Hlavná slučka programu *step()*
 - počíta, koľko času ubehlo od predch. snímky: *dt*
 - posunie a vykreslí všetky objekty v scéne (simulácia)
 - vyžiada si vytvorenie anim. rámca (volá samú seba)

```
function step() {  
  console.log("Step")  
  
  // Get time delta  
  var now = Date.now()  
  var dt = (now - time) / 100  
  time = now  
  
  move(dt)  
  draw()  
  
  requestAnimationFrame(step)  
}
```


View a Controller

```
// View
function draw() {
  // Clear canvas
  ctx.fillStyle = "white"
  ctx.fillRect(0, 0, canvas.width, canvas.height)

  // Render all objects in scene
  for (i in scene) {
    scene[i].draw()
  }
}
```

```
// Controller
function move(dt) {

  // Move all objects in scene (pass delta time)
  for (var i in scene) {
    scene[i].move(dt)
  }
}
```

Objekt *Ball*

- V konštruktore nastavíme atribúty na náhodné veličiny

```
class Ball {  
  // Initialization  
  constructor() {  
    this.image = document.getElementById("image")  
  
    this.x = Math.random() * canvas.width  
    this.y = Math.random() * canvas.height  
    this.dx = Math.random() * 50 - 25  
    this.dy = Math.random() * 50 - 25  
    this.size = Math.random() + .3  
    this.rotation = 0  
  }  
  
  // Movement logic  
  move(dt) { ... }  
  
  // Render self  
  draw() { ... }  
}
```

Pohyb objektu *Ball*

```
// Movement logic
move(dt) {
  console.log("hello")
  if (this.x > canvas.width) {
    this.x = canvas.width
    this.dx = -Math.abs(this.dx)
  }
  if (this.x < 0) { ... }
  if (this.y > canvas.height) {
    this.y = canvas.height
    this.dy = -Math.abs(this.dy) * 0.9 // Reduce movement
  }
  if (this.y < 0) { ... }

  // Movement - !!! each motion (speed, gravity vector) is multiplied with
  dt !!!
  this.x += this.dx * dt
  this.y += this.dy * dt

  this.dy += 9.8 * dt // Add gravitational force

  this.rotation += dt // Rotate based on time
}
```

Vykreslenie objektu *Ball*

- Transformácie

- aplikujú sa v opačnom poradí
- sa kumulujú, preto **každý** objekt by si mal najprv odložiť stav *Context*-u, a po vykreslení kontext obnoviť

```
// Render self - saving state allows to transform object
individually
draw() {
    ctx.save()           // Save current state
    ctx.translate(this.x, this.y)
    ctx.rotate(this.rotation)
    ctx.scale(this.size, this.size)
    ctx.drawImage(this.image, -20, -20, 40, 40)
    ctx.restore()        // Restore state
}
```

Balls

- Pozri implementáciu v ardesári 4_balls

Ďakujem za pozornosť