# Kurs języka Object/Delphi Pascal

## na bazie implementacji Free Pascal.

AUTOR ŁUKASZ STAFINIAK

*Email:* lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
*Web:* www.ii.uni.wroc.pl/~lukstafi

Jeśli zauważysz błędy na slajdach, proszę daj znać!

# Wykład 6: Dziedziczenie.

## Ray Tracing. Własności: property. Interfejsy.

# Wprowadzenie do Ray Tracingu

- *Ray-tracing* to metoda generowania grafiki poprzez śledzenie promieni światła wstecz, od kamery do obiektów na scenie, i dalej, modelując rozproszenie lub odbicie, do źródeł światła lub innych obiektów.
- Promień to półprosta zaczepiona w ustalonym punkcie i biegnąca w ustalonym kierunku.
  - Punkt i kierunek to wektory (trójki liczb). Dodatkowo będziemy pamiętać żeby kierunek był unormowany (tzn. długości 1).
- Potrzebujemy metody znajdującej przecięcie promienia i obiektu, tzn. punkt padania promienia na obiekt.
  - W typie `THit` dodatkowo pamiętamy parametry powierzchni w punkcie padania promienia:
    - `Distance` to odległość od początku promienia do punktu padania, `Infinity` oznacza brak przecięcia,
    - `Normal` to unormowany wektor prostopadły do powierzchni w punkcie padania, wskazujący na zewnątrz,
    - `ObjMat` to materiał powierzchni w punkcie padania.

- Scenę będziemy pamiętać jako *scene graph*, hierarchiczną strukturę przyspieszającą znajdowanie obiektów na trasie promienia.
  - W liściach drzewa-grafu sceny będą materialne obiekty, a w węzłach wewnętrznych będziemy pamiętać *bounding volume* (w przykładzie `BoundingBall`), czyli obiekt w którego środku są wszystkie obiekty poniżej.
- Przecięcie promienia $\vec{O} + t\,\vec{D}$ (zaczepionego w $\vec{O}$, biegnącego w kierunku $\vec{D}$ po $t$ dodatnich) ze sferą $(\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) - R^2 = 0$ znajdziemy rozwiązując

$$(\vec{O} + t\,\vec{D} - \vec{C}) \cdot (\vec{O} + t\,\vec{D} - \vec{C}) - R^2 \;=\; 0$$

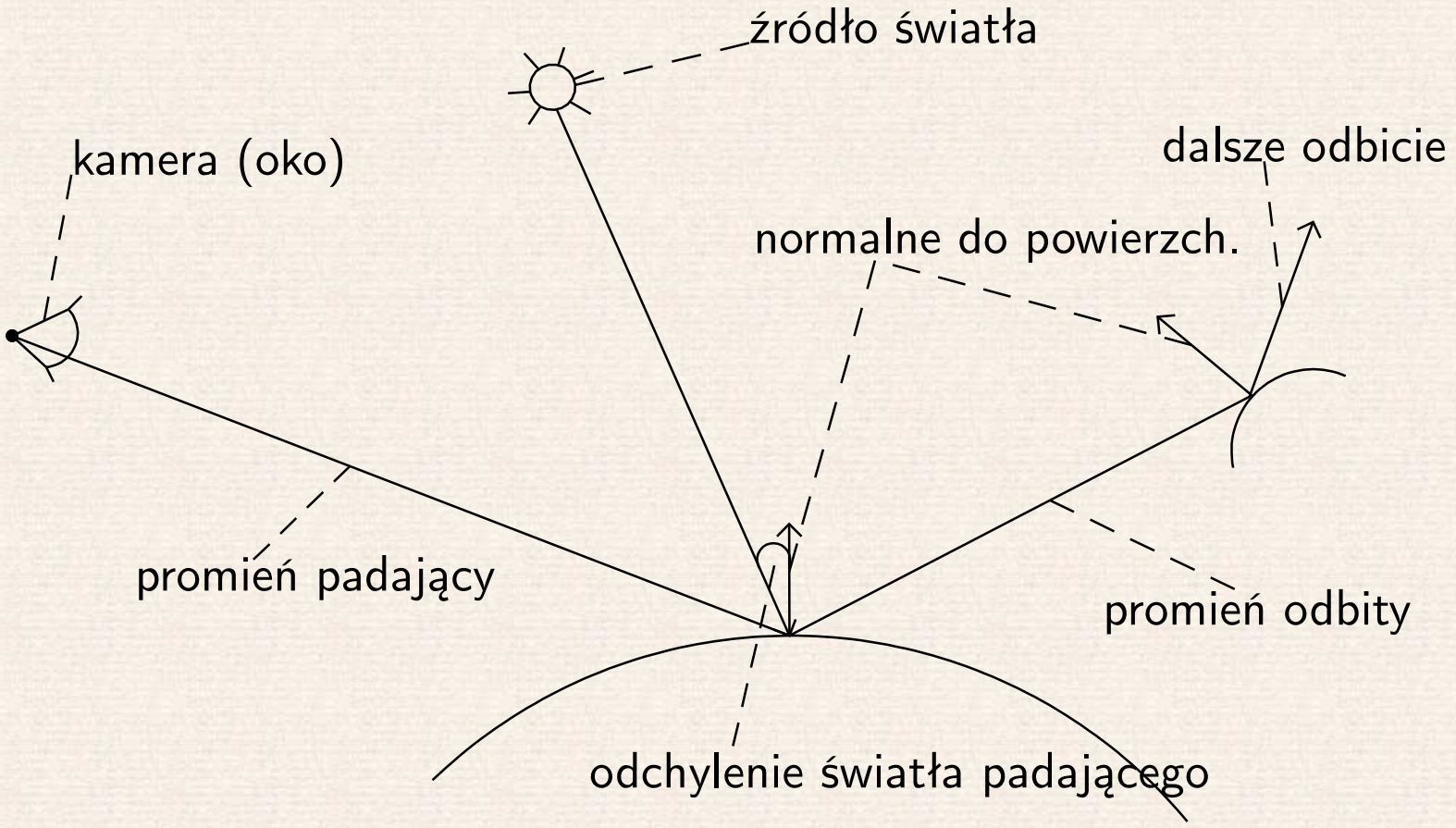
  ze względu na $t$. Dostaniemy równanie kwadratowe: wybieramy mniejsze z rozwiązań dodatnich (bliższy z maks. dwóch punktów przecięcia) – jeśli nie ma rozwiązań lub rozwiązania są ujemne, sfera nie jest na trasie promienia.
  - Unormowany kierunek, tzn. $\vec{D} \cdot \vec{D} = 1$, upraszcza obliczenia.
  - Wektor normalny to $\frac{\vec{P} - \vec{C}}{|\vec{P} - \vec{C}|}$ gdzie $\vec{P} = \vec{O} + t_d \vec{D}$ i $t_d =$ `Distance`.

- Przecięcie promienia z pudełkiem (prostopadłościanem) o krawędziach równoległych do osi układu współrzędnych, znajdziemy biorąc część wspólną odcinków promienia przechodzących przez trzy „płyty" ograniczające pudełko rozpięte między $(x_0, y_0, z_0)$ a $(x_1, y_1, z_1)$: $\left\{\vec{P} \middle| P_x \in [x_0, x_1]\right\}$ itd. – i zwracając bliższy z dodatnich końców (jeśli istnieje).
  - Rozwiązujemy:

$$\begin{aligned} x_0 &= O_x + t_{x_0} D_x \\ x_1 &= O_x + t_{x_1} D_x \end{aligned}$$

    i tak samo dla $y$, $z$. Dostaniemy odcinek: wzdłuż $\vec{O} + t\ \vec{D}$ od $t = \max\{\min(t_{x_0}, t_{x_1}), \min(t_{y_0}, t_{y_1}), \min(t_{z_0}, t_{z_1})\}$ do $t = \min\{\max(t_{x_0}, t_{x_1}), \max(t_{y_0}, t_{y_1}), \max(t_{z_0}, t_{z_1})\}$, o ile pierwsza liczba jest mniejsza od drugiej.

- Po znalezieniu punktu padania, liczymy natężenie światła z tego punktu jako kombinację dwóch efektów:
  - Dla powierzchni matowych, prawo Lamberta mówi że jasność obiektu jest proporcjonalna do kąta padania światła na powierzchnię (dokładniej, do cosinusa odchylenia kierunku światła od normalnej), bez względu na kąt patrzenia na powierzchnię.
    - Uwzględniamy ten składnik tylko jeśli źródło światła nie jest przesłonięte.
    - Dla efektu światła nieskończenie daleko (jak słoneczne), możemy uprościć biorąc zawsze ten sam kierunek światła.
  - Dla powierzchni lustrzanych, znajdujemy promień odbicia zgodnie z regułą „kąt padania jest równy kątowi odbicia", i rekurencyjnie bierzemy jasność punktu padania odbitego promienia.
- Obrazek budujemy z jasności punktów padania promieni puszczonych poprzez każdy piksel „ekranu" pomiędzy kamerą (okiem) a sceną.


źródło światła
kamera (oko)
dalsze odbicie
normalne do powierzch.
promień padający
promień odbity
odchylenie światła padającego

# Dziedziczenie

```
type                                  Klasy deklarujemy zwykle w interfejsie – inaczej niewidoczne na zewnątrz.
   TBaseClass = class
   strict private                     Część dostępna tylko w innych metodach tej klasy.
      HandleWithCare : Integer;       Pola muszą poprzedzać metody...
      procedure Boom (When : Integer);
   private                            Część dostępna w całym module (pliku).
      DataField : String;             ...ale tylko w danym bloku dostępności.
      function Auxiliary : String;
   protected                          Część dostępna też w klasach dziedziczących
      procedure SetData (Index : Integer; Data : String);
   public                             Część ogólnie dostępna.
      destructor Destroy; override;   Zastępujemy destruktor z TObject.
      function Add (Data : String) : Integer; virtual;
   end;                               Metodę virtual będzie można zastąpić.
   TDerivClass = class (TBaseClass)   Domyślny dostęp public.
      function Add (Data : String) : Integer; override;
   end;                               Musimy jawnie zastępować (override) metody.
```

```
type
   TBaseClass = class end;                    Domyślnie dziedziczy z TObject.
   TDerivClass = class (TBaseClass);          Skrót na nic nie dodającą
                                                        klasę pochodną.
   TAbstractClass = class                     Klasa abstrakcyjna, bo ma:
      procedure Invariant (arg : TDerivClass);
                  virtual; abstract;          metodę bez implementacji.
      procedure NotReally; virtual;
   end;                                  Przy zastępowaniu metody, typy argumentów
   TConcreteClass = class (TAbstractClass)      muszą się pokrywać.
      procedure Invariant (arg : TDerivClass); override;
      procedure NotReally; reintroduce;      Ta metoda nie zastępuje
   end;                                  tej z klasy bazowej, „anuluje” wirtualność.
```

- Klasa może być zamknięta: `class sealed` co oznacza że nie można z niej dziedziczyć.

```
unit ClassExample;

interface

type
   TTestClass = class
   strict private
      procedure HelloPrivate;
   private
      Data : String;
      procedure HelloLocal;
   protected
      procedure HelloProtected;
   public
      constructor Create (AData : String);
      procedure Hello;
   end;
```

```
implementation

procedure TTestClass.HelloPrivate;
begin
   WriteLn ('TTestClass.HelloPrivate: ', Data);
end;

procedure TTestClass.HelloLocal;
begin
   WriteLn ('TTestClass.HelloLocal: ', Data);
end;

procedure TTestClass.HelloProtected;
begin
   WriteLn ('TTestClass.HelloProtected: ', Data);
end;

procedure TTestClass.Hello;
begin
   Write ('TTestClass.Hello: calling '); HelloPrivate;
   WriteLn ('continuing TTestClass.Hello: ', Data);
end;
```

```
constructor TTestClass.Create (AData : String);
begin
   Data := AData;
end;

var L : TTestClass;

begin
   L := TTestClass.Create ('Initializer');
   //L.HelloPrivate;
   L.HelloLocal;
   L.Destroy;
end.
```

```
program ClassExampleTest;
uses SysUtils, ClassExample;

type
   TDerivTest = class (TTestClass)
      procedure HelloExpose;
   end;

   procedure TDerivTest.HelloExpose;
   begin
      Write ('Exposed: '); HelloProtected;
   end;

var
   V : TDerivTest;
begin
   V := TDerivTest.Create ('First Data');
   //V.HelloLocal;
   V.Hello;
   V.HelloExpose;
   V.Destroy;
end.
```

# O przeciążaniu raz jeszcze

- We Free Pascalu procedury i funkcje z jednego modułu są domyślnie przeciążane.
- Ale żeby przeciążyć procedury należące do różnych modułów, muszą one być w obydwu modułach zadeklarowane z modyfikatorem wywołania `overload`.
  - Również gdy są to metody odpowiednio klasy bazowej i pochodnej!
  - W Delphi zawsze trzeba używać `overload` przy przeciążaniu.
- `virtual` / `abstract` / `override` poprzedza `overload` (razem precyzyjnie wyrażają rolę metody)

# Metody klasy

- Zwykłe metody zawsze wywoływane są na instancji klasy (tzn. na obiekcie) i mają dostęp do instancji poprzez referencję `self` (odpowiednik `this` z C++ i Javy).
- *Metody klasy* zadeklarowane przez `class procedure` / `class function` nie mają dostępu do `self` ani do pól i metod instancji, ale za to nie potrzebują instancji by ich używać: można też wywołać przez nazwę klasy.
  - Odpowiednik metod `static` z Javy.
  - Przydatne m.in. w połączeniu z typami generycznymi.

```
  TCharacter = class sealed
    class function GetNumericValue(AChar : UnicodeChar) :
Double; static; overload;
    class function GetNumericValue(const AString :
UnicodeString; AIndex : Integer) : Double; overload;
```

# Wywoływanie metody z klasy bazowej

- Gdy zastępujemy metodę z klasy bazowej, często potrzebujemy w nowej metodzie wywołać starą.
  - Szczególnie w konstruktorze i destruktorze!
- Słowo kluczowe `inherited;` – bez parametrów – wywoła metodę którą zastępujemy, przekazując jej argumenty aktualnej metody.
  - Działa dla wszystkich metod, nie tylko konstruktorów.
  - Zwróć uwagę, że zastąpić można tylko metodę o tej samej sygnaturze.
- Żeby wywołać dowolną inną metodę `Method` z klasy bazowej:
  `inherited Method (arg);`

# Własności instancji: `property`

- Własności to specyficzne metody odczytu/zapisu: używamy nazwy tak jak pola instancji, ale w rzeczywistości odczyt będzie wywołaniem funkcji zadeklarowanej jako specyfikator odczytu (`read`) dla własności, a zapis wywołaniem procedury – specyfikatora zapisu (`write`).
  - Zamiast funkcji / procedury możemy podać faktycznie pola instancji w specyfikatorach, nie muszą się pokrywać.
  - Możemy pominąć jeden ze specyfikatorów dostając własność „read-only" lub „write-only".
- Własności indeksowane współdzielą funkcję/procedurę odczytu/zapisu.
  - Np. wygodne dla współrzędnych `X`, `Y`, `Z` indeksujących wektor.
- Własności tablicowe pozwalają wykorzystać składnię indeksowania tablicy `Obj.Item[I]` (dla własności `Item`). Indeksy są dowolnego typu!
- Własność domyślna `default` to własność tablicowa do której odwoła się `Obj[I]`.

## Wirtualne metody publiczne `TObject`

- `destructor Destroy;`
- `class function newinstance : tobject;`
- `procedure FreeInstance;`
- `procedure AfterConstruction;`
- `procedure BeforeDestruction;`
- `function Equals(Obj: TObject) : boolean;`
- `function GetHashCode: PtrInt;`
- `function ToString: ansistring;`

# Iteratory dla składni `for..in`

- Żeby móc używać `for...in` na instancjach klasy, powinna ona implementować interfejs:

```
IEnumerable = interface(IInterface)
  function GetEnumerator: IEnumerator;
end;
```

- `GetEnumerator` zwraca instancję klasy implementującej:

```
IEnumerator = interface(IInterface)
  function GetCurrent: TObject;
  function MoveNext: Boolean;
  procedure Reset;
  property Current: TObject read GetCurrent;
end;
```

- `IInterface`, także zwany `IUnknown`, to korzeń hierarchii interfejsów dla klas ze zliczaniem dowiązań – z standardową implementacją w `TInterfacedObject`.