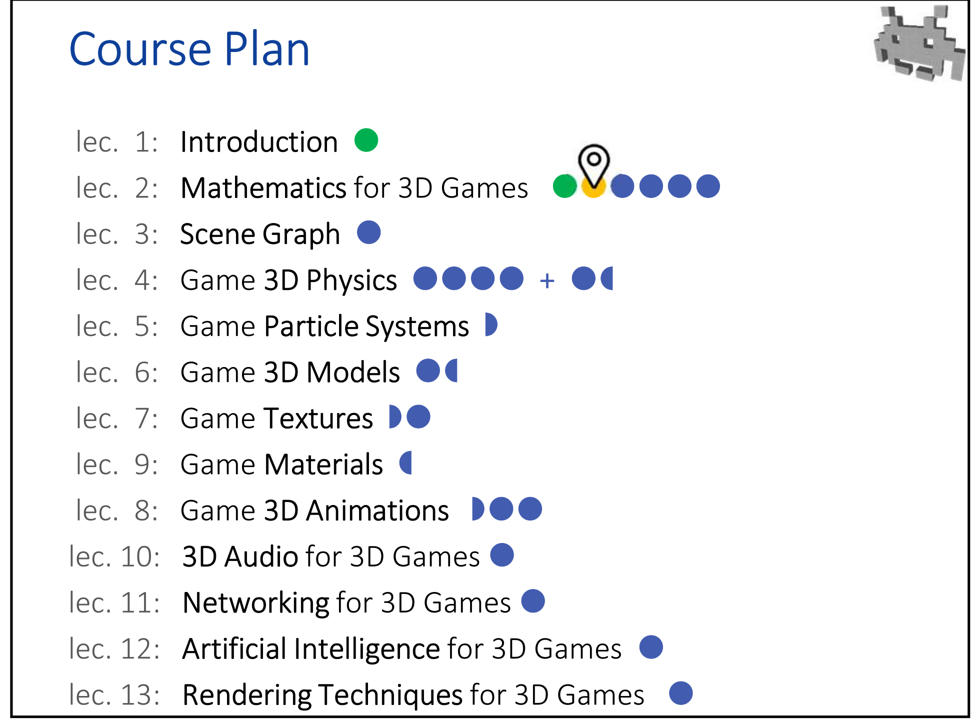
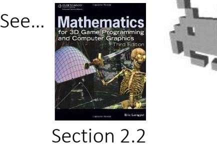


46

Point and vector algebra (summary 6/7)



Section 2.2

- Dot product (or inner product, or scalar product)
  - Output: a scalar
  - Alternative notations:

$$\vec{v} \cdot \vec{w}$$

$$\langle \vec{v} , \vec{w} \rangle$$

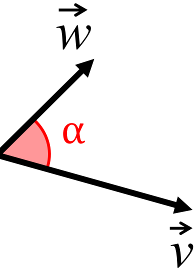
$$(v^{\mathrm{T}} \, w)$$

understand why

47

Point and vector algebra (summary 6/7)

- Dot product (or inner product, or scalar product)



$$\vec{v} \cdot \vec{w} = \|\vec{v}\| \; \|\vec{w}\| \; \cos(\alpha)$$

48

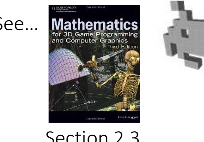
Point and vector algebra (summary 6/7)

- Dot product, useful to:
  - dot is zero: vectors are orthogonal (or, either vector is degenerate)
  - dot is positive: acute angle
  - dot is negative: obtuse angle

    applies to both vectors & versors
  - vector dot versor: extension of vector along direction;
  - vector dot versor: in other words, extract the coordinate of vector along that axis
  - versor dot versor: cosine of angle between them
  - versor dot versor: also, a similarity measure (in -1 +1)
  - any vector dot itself: its squared length

49

Point and vector algebra (summary 7/7)

See...


Section 2.3

- Cross product.
  Computed as:

$$\vec{v} \times \vec{w} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \times \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \begin{pmatrix} v_y\ w_z - v_z\ w_y \\ v_z\ w_x - v_x\ w_z \\ v_x\ w_y - v_y\ w_x \end{pmatrix}$$

50

Point and vector algebra (summary 7/7)

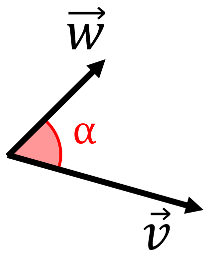
- Cross product, useful to:
  - find a vector orthogonal to two given vectors (*)
  - therefore: construct orthonormal basis
  - collinearity test (if colinear, then result is (0,0,0))
  - find (double) area of a triangle (floating anywhere in 3D)
  - find normal of a triangle in 3D (remember to renormalize it)
  - norm of (versor × versor): sin of angle

(*) a tip:

- dot product is to *TEST FOR* orthogonality (or to impose it)
- cross product is to *CONSTRUCT* orthogonality

51

Products and angles



$$\vec{v} \cdot \vec{w} = \|\vec{v}\| \; \|\vec{w}\| \; \cos(\alpha)$$

$$\|\vec{v} \times \vec{w}\| = \|\vec{v}\| \; \|\vec{w}\| \; \sin(\alpha)$$

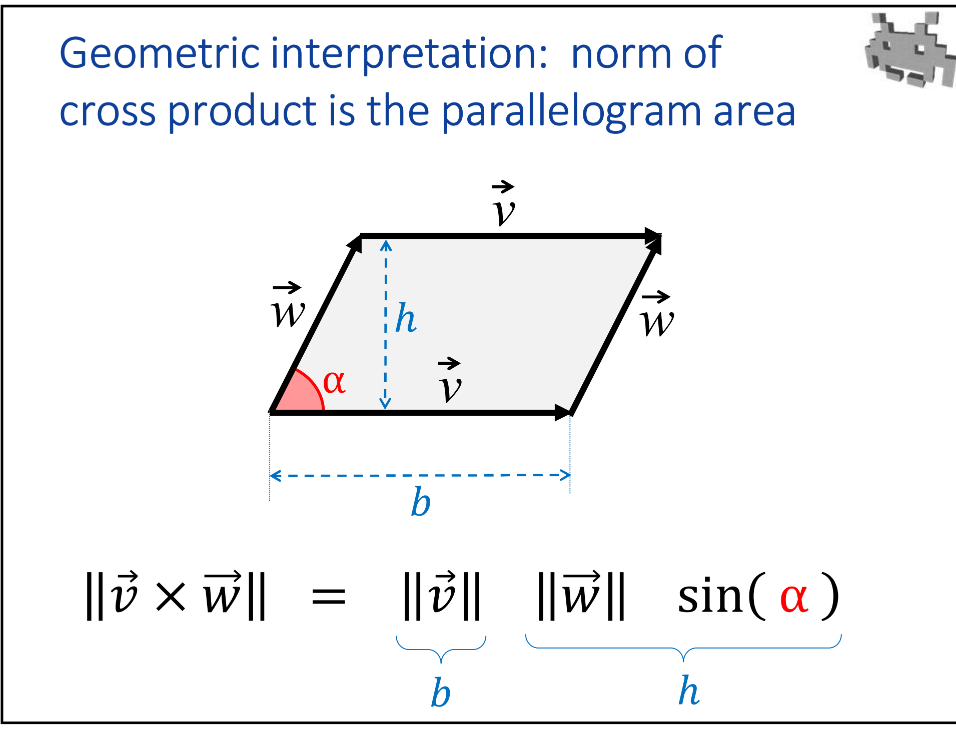
52

Cross product: complete geometric description

$$\vec{u} = \vec{v} \times \vec{w}$$

- Length of $\vec{u}$ = $\|\vec{v}\| \; \|\vec{w}\| \; \sin(\alpha)$
- Direction of $\vec{u}$ = orthogonal to both $\vec{v}$ and $\vec{w}$
- Verse of $\vec{u}$ = use the «right-hand rule» or the «left-hand rule»
  - whicever hand you are using to imagine your vector space! (and reference frame)

53

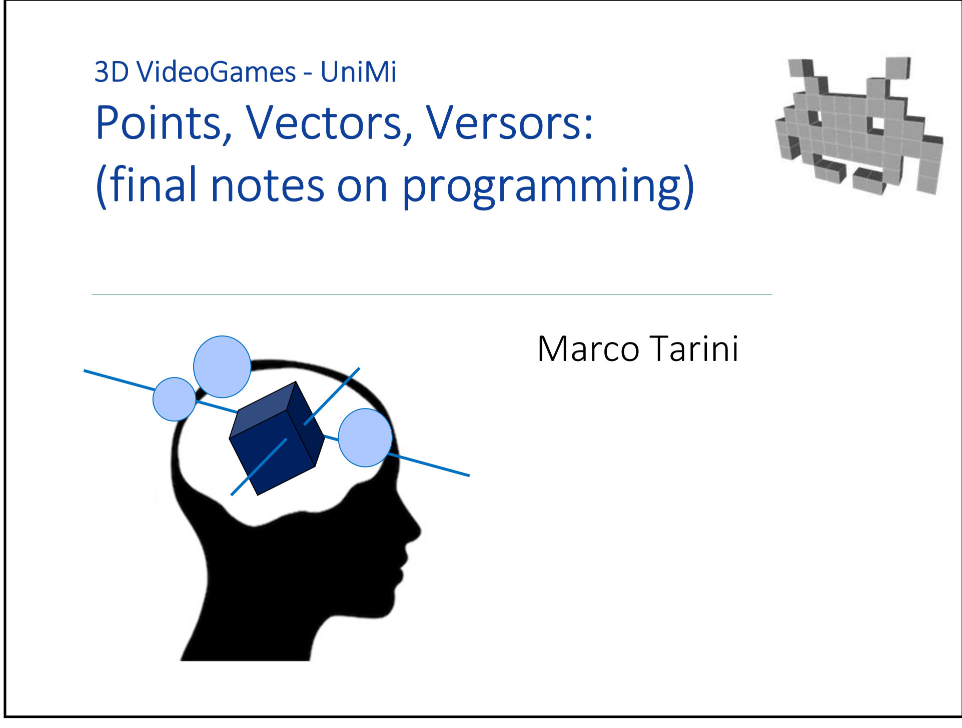


54

A note: generalization to N dimensions

- Everything we see (so far) for 3D points/vectors/versors immediately generalizes in 2D (for 2D games!), or even in N>3 dimensions
  - Quickly verify!
- With one exception: the cross product is only defined in 3D
  - But in 2D, the problem of constructing a vector/versor orthogonal to one given vector/versor (just one! Not two, like in 3D) is easy: *"swap coordinates, flip one* sign"*
  - That is: (x,y) is orthogonal to $(-y,x)$, and also to $(y,-x)$ -- verify with dot!
  - *: which coordinate you flip determines if you rotate 90° clockwise or counterclockwise -- verify with a drawing!
  - What happens if you extend with z=0 and apply the 3D cross? Try it with math! You'll find a 2D analogous of cross product: vector × vector = scalar (a useful operation!)

55



56

Points, Vectors, Versors: Internal representation

- *n*-tuple of scalar values (*n* is the dimension)
  - for us (usually): $n = 3$ (at times, 2 or 4)
  - they are the Cartesian coordinates of the point/vector
  - e.g.:

```
class Vector3 {
  // fields:
  float coords[3];

  // methods:
  …
}
```

or:

```
class Vector3 {
  // fields:
  float x, y, z;

  // methods:
  …
}
```

- note: the same structure is often used to represent points, vectors, and versors

59

Caveat (about coding): one type, multiple semantics

- Many libraries/engines/languages opt to use the same data type for 3D points, 3D vectors, 3D versors, (plus, sometimes: colors, and more)
  - alternatively, a library can use different types, e.g. Vector, Point, Versor
- Still, they should not be considered the same thing
  - that's nothing new: likewise, we use the same scalar data types ("float", "doubles") with widely different semantics (e.g. "weight", "volume", "temperature"…).
- It is up to the coder to *operate* on them accordingly
  - e.g.: it's not ok to sum a *temperature* with a *surface area*
  - e.g.: it's ok to divide a *weight* by a *volume* (and get a *specific weight*)
- which operations do make sense on points, vectors, versors?
  - the ones we have seen in their *algebra* !

60

Points, Vectors, Versors: Internal representation

- same class for points, vectors, and versors
- this is done in many libs & languages, e.g.:

class Vector3
https://docs.unity3d.com/ScriptReference/Vector3.html

class FVector
http://api.unrealengine.com/INT/API/Runtime/Core/Math/FVector/

- and also: GLSL, HLSL, GLM, Eigen, VcgLib , three.js, …

(GLSL, HLSL, GLM: vec3; Eigen: Vector3d; VcgLib: Point3d; three.js: Vector3)
(GLSL, HLSL: shader languages; GLM, Eigen, VcgLib: C++ libraries; three.js: JavaScript library)

61

Classes for Points / Vectors / Versors
A few examples in C++ libraries

- GLM library (*Graphics*) : class `vec3`
- UNREAL ENGINE library (*Videogames*) : class `VectorF`
- Eigen lib (*Linear Algebra*) : class `Vector3d`
- VCG-Lib (*Geometry processing*) : class `Point3f`
- Point Cloud Lib (*Geometry Processing*) : class `ON_3dVector`
- openMesh for (*Geometry processing*) : class `VectorT`
- cgall for (*Geometry Processing*) : class `Vector3`
- CinoLib (*Geometry Processing*) : class `vec3d`
- OpenCV for (*Computer Vision*) : class `Point3f`
- bullet for (*Physical Simulation*) : class `btVector3`
- ODE for (*Physical Simulation*) : class `dVector3`

62

Classes for Points / Vectors / Versors:
Other examples in C++ like languages

- GLSL shader language from Chronos : type `vec3`
- HLSL shader language from DirectX : type `float3`
- unity , C# *(videogames)*: class `Vector3`
- three.js , JavaScript *(graphics)* : class `Vector3`

63

Programming with vector algebra: the code looks like the expressions

- Concept ("on paper"): $\mathbf{p} = \mathbf{p} + k\,\hat{\mathbf{d}}$
- Code:
  - Data types:

```
Point3D dragonPos = …;
Versor3D dir = …;
float k = …;
```

  - Beginner's style code:

```
dragonPos.x = dragonPos.x + dir.x * k ;
dragonPos.y = dragonPos.y + dir.y * k ;
dragonPos.z = dragonPos.z + dir.z * k ;
```

  - What you should do:

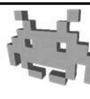
```
dragonPos.add( dir.scale(k) );
```

or (depending on the language)

```
dragonPos += dir * k ;
```

64

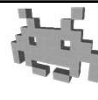
Pro-tip: try making your code assumption free!



personal opinion: the most standard one, among 3D modellers too

- Unity: left-handed: X-right, Y-up, Z-forward
- Unreal: left-handed: X-forward, Y-right, Z-up
- 3ds-Max: right-handed, Z-up
- Blender: left-handed, Z-up
- most VR systems: right-handed, Y-up
- OpenGL: (clip space) right-handed, Y-up
- DirectX: (clip space) left-handed, Y-down

65



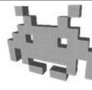
Pro-tip: try making your code assumption free!

E.g.: to move a pos 2.5 units “to the right”:

```
Vector3 pos = new Vector3 ( … );

pos.x = pos.x + 2.5;   // maybe ??
pos.y = pos.y + 2.5;   // hmm…??
```

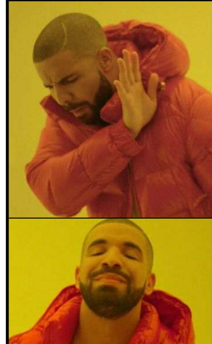
```
Vector3 pos = new Vector3 ( … );

pos += Vector3.right * 2.5;
```

67



Pro-tip: try making your code assumption free!

E.g.: to move a pos 2.5 units “to the right”:



```
FVector pos = FVector( … );

pos.X += 2.5f;   // maybe ??
pos.Y += 2.5f;   // hmm…??
```

```
FVector pos ( … );

pos += FVector::RightVector * 2.5f;
```

68