Computer Graphics I

- An Overview -

Sandro Spina

Computer Graphics and Simulation Group

Computer Science Department

University of Malta

Welcome

- The course is being organised by CGSG.
- Course Outline :
  - Mathematics : Some Linear Algebra + Trigonometry
  - Illumination (Models, Rendering Eqn, Physically based)
  - Viewer Optics
  - Geometry Transformation Pipeline
  - Texturing, Sampling and Filtering
  - Advanced Texturing (TBN Matrix)
  - GPUs and GPGPUs
  - Scene Representation
  - Individual Assignment: Game Design.

Books & Software

- Real-Time Rendering 3rd Edition (Tomas Akenine-Moller, Eric Hanies)
- Computer Graphics 2nd Edition (Foley, vam Dam, Feiner, Hughes)
- Mathematics for 3D Game Programming and Computer Graphics (Eric Lengyel)

- Any IDE which supports C# and XNA
- Recommended: Microsoft's Express Edition for C# or the XNA Development Centre (free editions)
- Other programming languages may be used …

By the end of the course …

- You should be able to appreciate what the process of creating 3D applications involves … because
- You should be able yourselves to write software which takes advantage of 3D.
- The ability to create 3D enabled applications will give you an edge in any future software you will build.
- Software: C# using XNA Framework

Real-time Rendering …

- Before we start: these slides are heavily based on one of the best books out there on real-time computer graphics.
- According to Tomas A Mueller (and many others):
  - Real-time rendering is concerned with the ‘making’ of images rapidly on the computer.
- As such it is the most highly interaction field of computer graphics (others include offline rendering, CV, etc)
- An image appears on the screen, the viewer reacts to it, and this feedback effects what happens next.
- All this happens many times per second … These still images become animations.

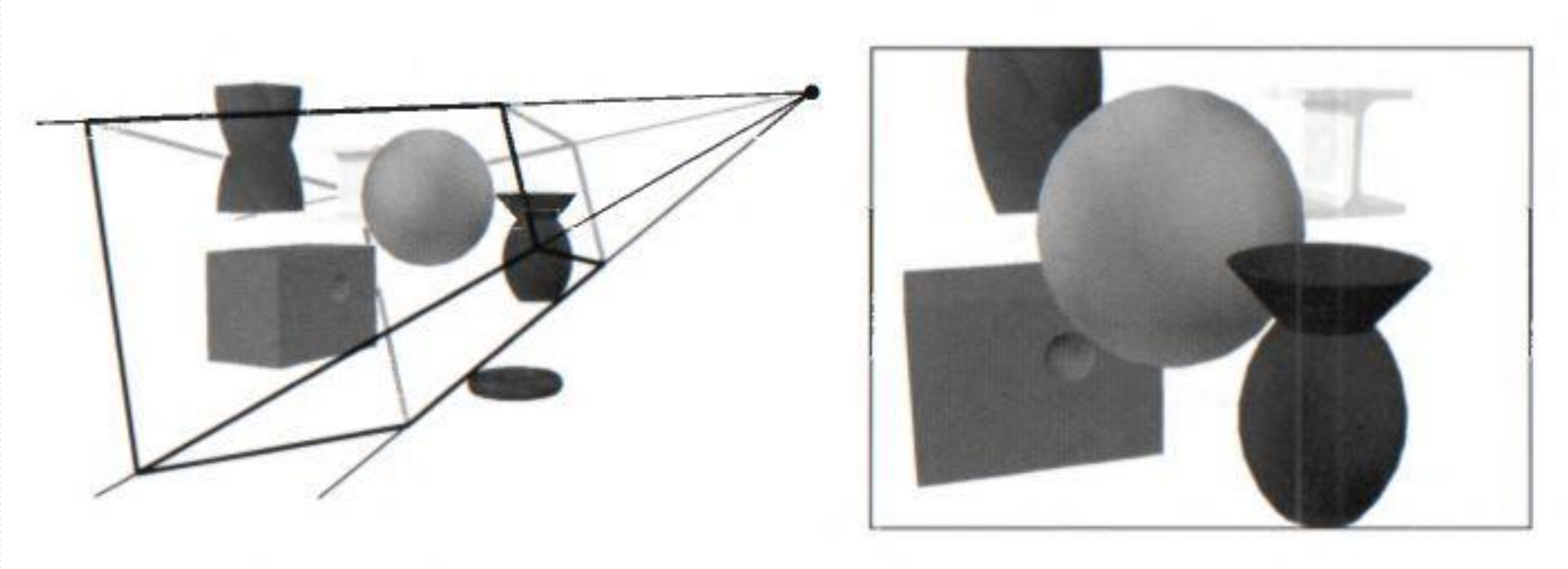
Real-time Rendering …

- Frames Per Second: From 6fps to 72fps
- Rendering in real-time … In 3D
- Graphics Acceleration Hardware
- Gaming (in general) is probably the most prominent area in which real-time rendering is made use of.
- PS3, XBox360, Wii, DS/i, iPhone, PSP, many mobile devices, are now embedding some form of graphics chip which is able to perform real-time rendering.
- The market for real-time rendering (on everything) is now booming …. evolutions in hardware is making unbelievable advances.

The Graphics Rendering Pipeline

- The main role of the pipeline is to generate (render) a two-dimensional image, given:
  - A virtual camera
  - Three-dimensional objects
  - Light sources,
  - Shading equations,
  - Possibly other components of the 3D environment.
- So this pipeline is fundamental for rendering and is thus the underlying tool for real-time rendering.
- Different Hardware / software might have a different (slightly perhaps) pipeline but at the end of it if you want to render something it has to pass through this pipeline.

RTR Pg 12 – Frame Render



Pipeline Stages – Architecture

- Every stages of the pipeline prepares the data for the next stage. This is true for all pipelines in factories etc ...

- The speed of the pipeline is determined by the slowest stage ... which then determines the frames per second (together with other things such as the amount of data processed)

- The pipeline stages execute in parallel (that is why the previous point is true of course).

- A high-level conceptualisation of the rendering would be as split in the following three stages:
  - Application
  - Geometry
  - Rasterizer

Highlevel Pipeline Stages – Diagram

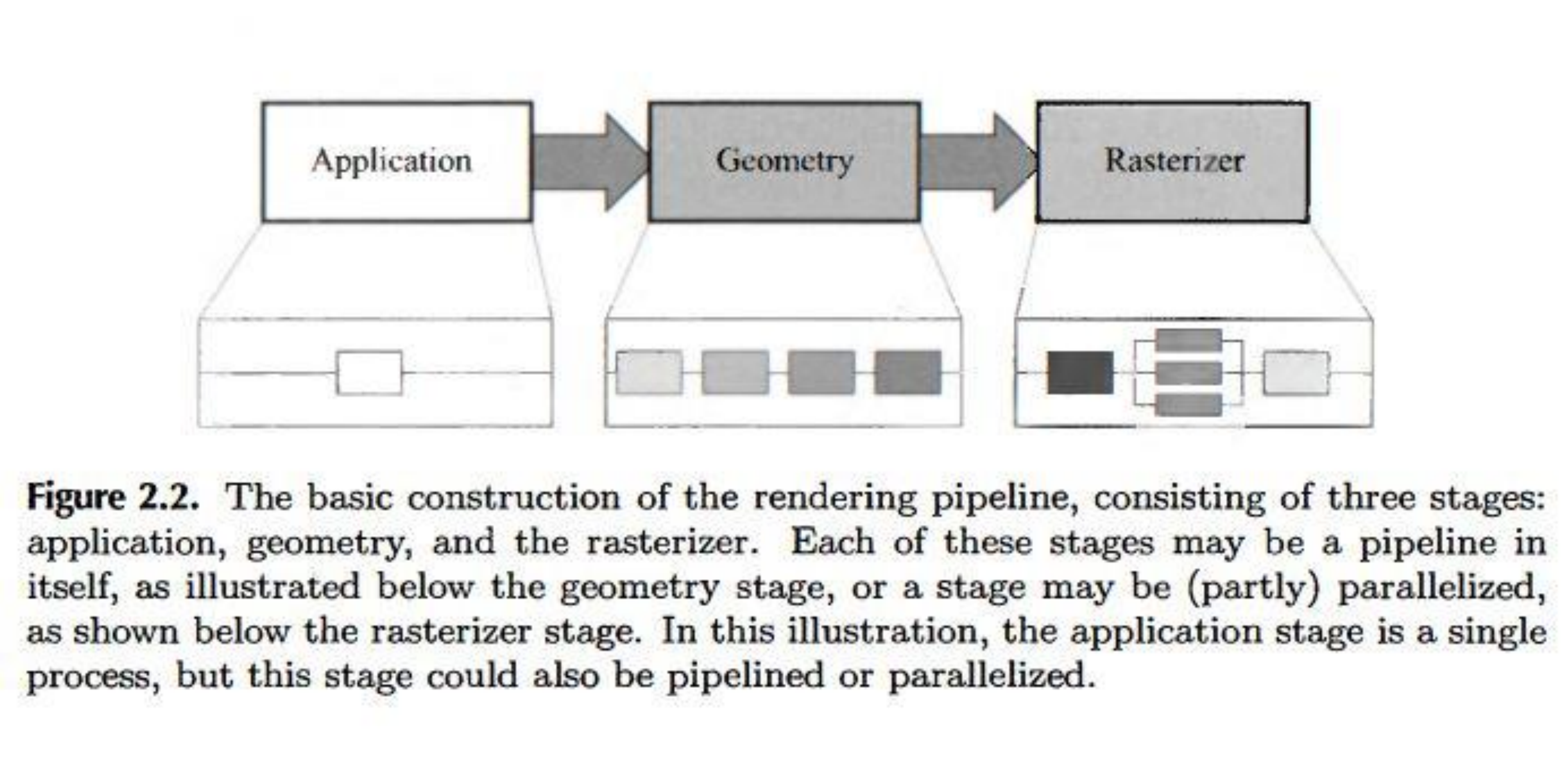


Figure 2.2. The basic construction of the rendering pipeline, consisting of three stages: application, geometry, and the rasterizer. Each of these stages may be a pipeline in itself, as illustrated below the geometry stage, or a stage may be (partly) parallelized, as shown below the rasterizer stage. In this illustration, the application stage is a single process, but this stage could also be pipelined or parallelized.

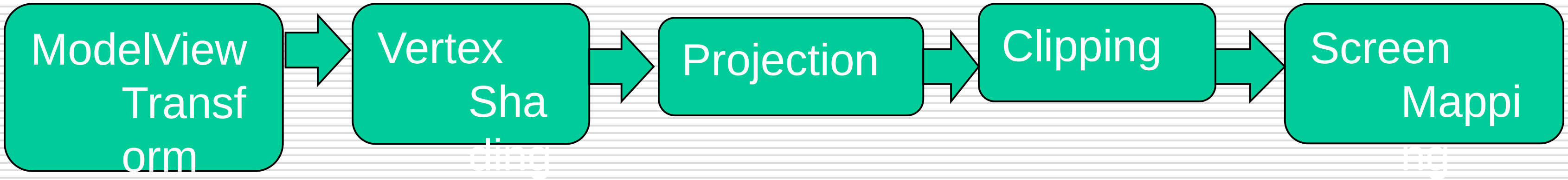
The Application Stage

- This is the initial stage .... carried out on the CPU which feeds (with data – geometry primitives) the GPU pipeline.
- The developer has full control over what happens in the application stage.
- Optimisations in this stage effect the entire pipeline ... For example Level of Detail calculations.
- Consider the game of pacman and his chase for the ghosts. In this stage of the pipeline the programmer handles one very important task ... Collision detection (has the geometry of the ghost intersected the geometry of pacman)

The Geometry Stage

- Primarily responsible for the majority of per-polygon and per-vertex operations.

- Further subdivided into :
  - Model and view Transforms
  - Vertex shading
  - Projection
  - Clipping
  - Screen Mapping



ModelView Transf orm

Vertex Sha ding

Projection

Clipping

Screen Mappi ng

Model and View Transform

- Different Coordinate Systems / Spaces
  - Object Model Space (coordinates are relative to an origin in the object itself)
  - A model transform is then associated with the model which essentially moves it to a particular space in world coordinates.
  - The model is now in world space coordinates.
  - Various instances can be created of the same object by applying different model transformations to the same model.
- The world space is unique … There's only one of it. All models to be rendered are in this space now.
- We now transform everything from world space into camera space … i.e. Relative to the location from where the camera is pointing. Note that camera also has a location in world space.

Vertex Shading

- So far we've looked into the geometric aspects (shape + position)

- At this stage we also need to model the appearance of the objects.

- Appearance relates to the material of the object + the interactions between light and the material.

- Shading:
  - "The process of determining the effect of lighting on a material"

- Uses a shading equation … Which at this stage performs per vertex computations. These values are then passed to the rasterisation stage.

Projection …

- After vertex shading ... Our models (in camera/eye space) need to be projected onto a plane (3D->2D transform in theory).

- More formally the projection transforms the view volume into a unit cube with its extreme points at (-1,-1,-1) and (1,1,1) called the *canonical view volume*.

- Orthographic Projection (Preserves parallel lines)
- Perspective Projection (essentially how we see things, the further they are from the camera the smaller they become)

- Note that here we are really transforming the view volume (frustum or rectangular box volume) into a normalised canonical view (another volume). It is still a projection because after display the z-coordinate is not stored in the image generated. Stored in Z-Buffer for depth calculations.

Clipping (i)

- We only need to shade the objects (models) which are within the field of view of the camera.

- After projection these objects will be those objects which are in the normalised view volume.

- Only these primitives wholly (or partially) inside the view volume need to be passed to the rasterizer stage, which then draws them on the screen. Anything else is a waste of computational resources.

- Clipping needs to be carried out on those primitives that are partially within the view volume.

- View transformation and projection are carried out before in order to make this step easier. Remember that we can now clip against a unit view volume.

Clipping (ii)

- Clipping against 6 planes ...

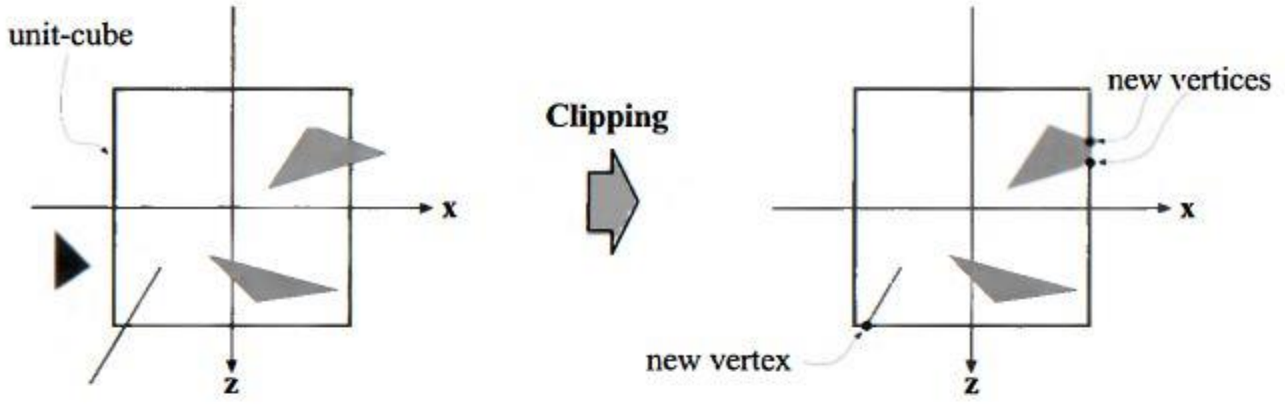


Figure 2.6. After the projection transform, only the primitives inside the unit cube (which correspond to primitives inside the view frustum) are needed for continued processing. Therefore, the primitives outside the unit cube are discarded and primitives totally inside are kept. Primitives intersecting with the unit cube are clipped against the unit cube, and thus new vertices are generated and old ones are discarded.

Screen Mapping

- After clipping, primitives are now passed to the screen mapping stage.
- The coordinates are still 3 dimensional (even after projection) at this stage ... But only the x and y coordinates (of each primitive) are used and transformed to screen coordinates.
- Screen coordinates + z-coordinate = Window Coordinates
- Screen coordinates are obtained by translating and scaling (non-uniform scaling most of the times)

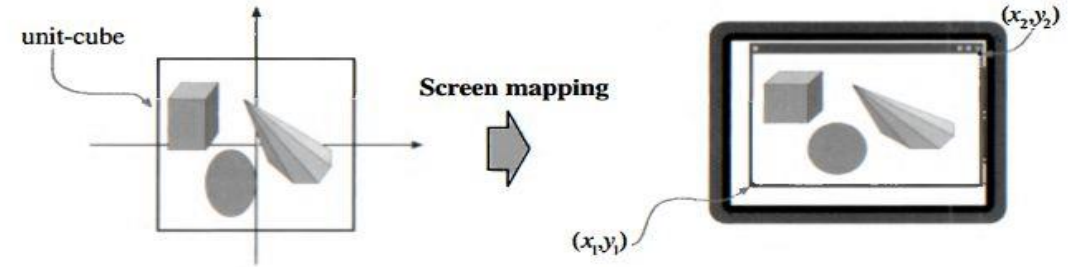


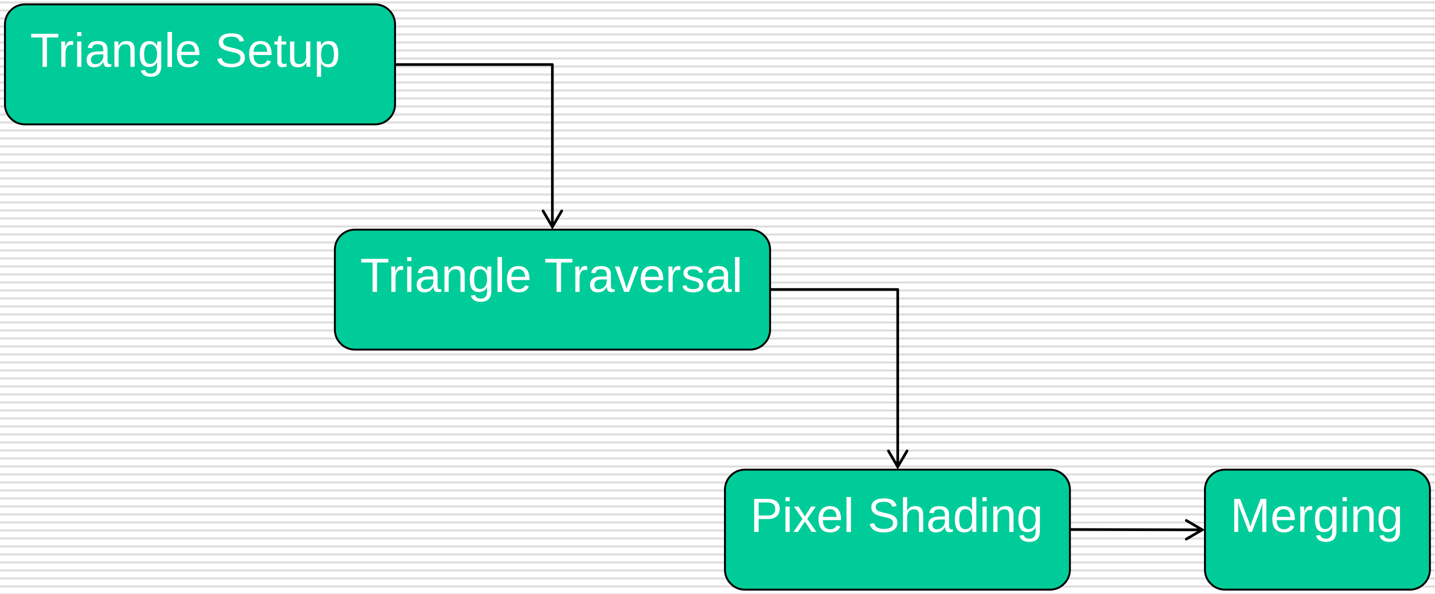
Figure 2.7. The primitives lie in the unit cube after the projection transform, and the screen mapping procedure takes care of finding the coordinates on the screen.

The Rasterizer Stage

- Final stage in our high-level conceptualised pipeline.
- We now have transformed and projected vertices from all the primitives in the normalised view volume.
- The goal of this stage is to compute and set colours for the pixels covered by the object.
- This process is called rasterisation or scan conversion, which finally outputs the pixels on the screen.
- It's inputs (associated with each vertex) are:
  - 2D vertices in screen space
  - z coordinates representing depth
  - various shading information

The Rasterizer Stage Pipeline

- The rasteriser stage (like previous stages) is in itself also split into functional stages forming a pipeline.



Triangle Setup and Traversal

- Triangle setup is something fixed done by the hardware in which you can't do much.
- Essentially the triangles are prepared for shading
- Triangle Traversal is the process of finding which pixels are inside a triangle.
- Each triangle fragment's properties are generated using data interpolated among the three vertices.
- Once we establish which pixels are effected by the triangle being rendered ... Pixel shading is carried out.

Pixel Shading

- Every per pixel computation is performed here … i.e. All the pixels that fall within any of the fragments we are rendering.
- The end result is a colour for each pixel … which is further passed to the next (and final) stage of the pipeline.
- On today’s modern GPUs this part of the pipeline is completely programmable and is executed by the GPU shader cores.
- Texturing can be applied here … To increase the realism of a material.
- Pixel shaders
  - HLSL = Shader used with the DirectX API
  - GLSL = Shader used with the OpenGL API
  - CG = Generic shader language

Merging (i)

- Merging of different buffers in the system to output the final set of pixels forming the frame.
- Information for each pixel is stored in the *colour buffer*. Essentially a 2D array of colours.
- The merging stage updates this buffer with the information (fragment colour) received from the shading stage.
- This stage is not programmable but is highly configurable (similar to the fixed function pipeline)
- This stage also resolves visibility using (usually) the z-buffer algorithm.
- The algorithm makes use of a depth buffer which stores the z-coordinates from the projection stage.

Merging (ii)

- A *stencil buffer* can also be used in this stage if required by the effects which are currently active (example shadows).
- The *accumulation buffer* can be used to simulate effects like motion blur. Images can be accumulated over a number of frames using a st of operators.
- Other effects include simulating depth of field, antialiasing, soft shadows, etc.
- *Double Buffering* : Rendering takes place offline (while you are watching the previous frame on your monitor) in a *back buffer*.
- The content of the *back buffer* is swapped as soon as the scene is rendered with the contents in the *front buffer* during vertical retrace of the monitor.

Conclusions on the pipeline

- The pipeline which is described in Mueller's Real-Time Rendering book (summarised in these slides) has been evolving over a number of years over different hardware evolutions.
- This is not the only possible pipeline … Offline rendering for example undergoes a different process altogether.
- For many years we have lived with the fixed-function pipeline defined by the graphics API in use (OpenGL or Direct3D)
- The last (most probably) example of a fixed function pipeline if Nintendo Wii console. PS3 and XBOX360 are programmable.
- In most parts of this course we shall be using the programmable version of the pipeline …

The Game Engine (i)

- The graphics pipeline is responsible for the rendering of objects within our virtual world …
- But prior to this some processing needs to take place in order to setup this virtual world (scene)
- Although (as we shall see) this can be done by creating the geometries, processing them and then sending them to the pipeline this is usually done is a more structured (in terms of software design) way.
- This is were the *game engine* comes into effect.

The Game Engine (ii)

- In an interactive 3D application such as a game, a large amount of code is required to do all the actions required from the game.
- A typical Game Engine loop:
  - Draw current scene on the screen
  - Animate characters … (e.g. players running)
  - Detect collisions between objects in the scene … (e.g. in a FPS)
  - React to these collisions in a physically correct manner so that the animation are realistic (and convincing)

Assignment

- Constitutes 40% of your final mark.
- You will **NOT BE** divided into groups (individual effort required)
- You'll be developing a game-like application ... which we'll be discussing during class.
- You are encouraged to include in the development of the game your ideas
- Not Multiplayer (although you can look into this option if you want)
- Deliverables: Report (***VERY IMPORTANT***) + source & executable code.

Tools

- XNA Framework
- DirectX
- OpenGL – with CodeBlocks
- Blender (or any other modelling tool)
- Game Engines … For C++, Java, Python, etc ….
  - OGRE, JMonkey