CE318: Games Console Programming

Revision Lecture

Philipp Rohlfshagen

prohlf@essex.ac.uk
Office 2.529

Format of Exam

You have 2 hours for the exam. It contains 4 questions (all equal weight):

- Exam counts 70% of your final mark.
- All questions should be answered (no options).
- Questions may relate to any of the topics covered in the course.
- Answers to questions might not be in the lecture slides.

Format of today's lecture:

- First go through some important topics
- Then discuss topics you choose / Q&A

After today's lecture, email me if you have questions: prohlf@essex.ac.uk.
(don't leave it to the last minute).

Recap: Topics Covered in CE318

- Structure of an XNA 4.0 game
- The game loop
- Game efficiency and optimisation
- Game content/assets
- Game design
- Game physics
- 2D/3D game implementations
- Simple game AI
- 2D and 3D graphics
- Vectors, matrices & triangles
- Texturing, shading, lighting
- 3D model creation
- Camera types
- Multiplayer games

Topics for Today's Revision

1. C-Sharp Basics
2. The Game Loop
3. 2D Basics
4. 3D Basics
5. Models & Collision detection
6. A*
7. Behaviour Trees
8. Billboarding

Outline

The Basics of C#

```
using System;

namespace HelloWorld
{
  class Hello
  {
    static void Main(string[] args)
    {
      PrintHelloWorld();
      Console.WriteLine("Press any key to exit.");
      Console.ReadKey();
    }

    static void PrintHelloWorld()
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

The word `using` is used to import packages each of which is known as a `namespace`. You can have many classes in each namespace and the filename does not need to correspond to either the namespace nor any of the enclosed classes. Method names start with a capital letter, whereas the built-in types `class` and `string` do not.

The Basics of C#: Properties

Properties are shortcuts for the getters and setters usually associated with class variables.

```
private String Name;

public String Name
{
  get { return Name; }
  set { Name = value; }
}
```

An even shorter version is:

```
public String Name { get; set; }
```

Now the class has a property called `Name` which can be set by calling

```
Class.Name="My Name";
```

Use access modifiers (e.g., `private`) to prevent anyone from changing the value of the property.

The Basics of C#: Structs

Like a class but a value type. Also, they do not support inheritance.

```
struct Square
{
  public int Width { get; set; }
  public int Height { get; set; }

  public Rectangle(int width, int height)
  {
    Width = width;
    Height = height;
  }

  public Rectangle Add(Rectangle rect)
  {
    Rectangle newRect = new Rectangle();

    newRect.Width = Width + rect.Width;
    newRect.Height = Height + rect.Height;

    return newRect;
  }
}
```

Default constructor initialises all variables to 0; you cannot assign a value to them (unless they are `const`). Structs do not require instantiation of an object on the heap.

The Basics of C#: Enumerations

Enumerations are used frequently in games to indicate different states of the game (e.g., splash screen, paused, etc.):

```
public enum AnEnumeration { TYPE1, TYPE2, TYPE3, TYPE4 };
```

Enumerations are often used in conjunction with switch statements:

```
public void IllustrateEnumerations(AnEnumeration type)
{
  switch (type)
  {
    case AnEnumeration.TYPE1:
      Console.WriteLine("Type 1");
    break;
    case AnEnumeration.TYPE2:
      Console.WriteLine("Type 2");
    break;
    case AnEnumeration.TYPE3:
      Console.WriteLine("Type 3");
    break;
    case AnEnumeration.TYPE4:
      Console.WriteLine("Type 4");
    break;
  }
}
```

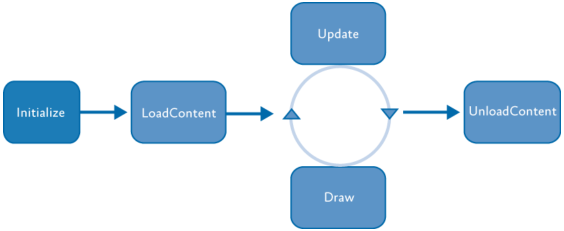
Each `enum` member has an integral value (following declaration order).

Outline

The Game Loop

XNA creates a template of the game class for you. In order to write a game, all one needs to do is to replace the `// TODO` comments with some game-specific code.

- `Constructor()` Constructor of the game
- `Initialize()` Initialise assets that do not require a GraphicsDevice
- `LoadContent()` Load game assets such as models and textures
- `UnloadContent()` Release game assets
- `Update()` Update the game's logic
- `Draw()` Render all game objects and backgrounds on the screen



Game Loop Timing

Following initialisation and loading content, XNA repeatedly calls the `Update()` and `Draw()` methods.

- A Game is either fixed step (default) or variable step
- The type of step determines how often `Update()` will be called
- A fixed-step Game tries to call `Update()` every `TargetElapsedTime`
- Default `TargetElapsedTime` is 1/60th of a second (60 fps)
- Game only calls `Update()` when its time but possibly skips call to `Draw()`
- If `Update()` takes too long, Game sets `IsRunningSlowly` to true
- You can check value of `IsRunningSlowly` in `Update()` to detect dropped frames
- You can reset the elapsed times by calling `ResetElapsedTime`

To ensure smooth animations (if expecting delays), take time passed since last call to update into account.

Game Loop Timing

XNA calls `Update()` and `Draw()` 60 times a second. This ensures there is no flickering on the monitor displaying the graphics. It is possible to change the frame rate in the constructor:

```
TargetElapsedTime = new TimeSpan(0, 0, 0, 0, 50);
```

This forces XNA to call `Update()` only very 50 milliseconds (i.e., 20 fps).

Many objects will depend on the number of times `Update()` is called. Can use *animation frames*:

```
int timeSinceLastFrame = 0;
int millisecondsPerFrame = 50;
```

These variables can be used to update an object only if `millisecondsPerFrame` have passed. You can get the time elapsed in between calls to `Update()` using `gameTime.ElapsedGameTime.Milliseconds`.

Outline

Drawing Sprites

The term *sprite* is used for images drawn on the screen. In `Game1`, the following lines are included in the class definition:

```
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;
```

The `GraphicsDeviceManager` allows access to the graphics device `GraphicsDevice` and the `SpriteBatch` does the actual drawing of the sprite.

Running `Game1` creates a frame with blue background:

```
GraphicsDevice.Clear(Color.CornflowerBlue);
```

In order to display an image, we need to

1. Add the image to the game's content pipeline
2. Load image into game
3. Draw the image

Drawing Sprites

Add the image:

1. Right-click on the content project header
2. Select *Add - Existing Item ...*
3. Choose file

We then add a variable for the image in `Game1`:

```
Texture2D image;
```

and load the image (in `Game1.LoadContent()`):

```
image = Content.Load<Texture2D>("image")
```

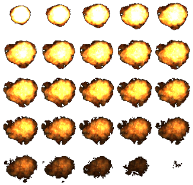
where "image" is the name of the file (extension is not required).

Finally, draw the image (in `Game1.Draw()`):

```
spriteBatch.begin();
spriteBatch.Draw(image, GraphicsDevice.Viewport.Bounds, Color.White);
spriteBatch.end();
```

2D Animations

A simple way to generate animated sprites is to use **sprite sheets**.



- A sequence of images on one sheet
- Images created in some DCC package
- Use source rectangle to choose image
- Move source rectangle across sheet
- Creates a moving image

```
Point frameSize = new Point(50, 50);  // individual image size
Point currentFrame = new Point(0, 0); // start top left
Point sheetSize = new Point(5, 5);    // rows and columns
```

In `Game1.Update()`, loop over sprite sheet, then draw:

```
spriteBatch.Begin();
spriteBatch.Draw(texture, Vector2.Zero, new Rectangle(currentFrame.X *
     frameSize.X, currentFrame.Y * frameSize.Y, frameSize.X,
    frameSize.Y), Color.White, 0, Vector2.Zero, 1, SpriteEffects.None
    , 0);
spriteBatch.End();
```

demo

Outline

What is 3D?

At the core of any 3D graphic are many triangles that define its shape. This is known as a **geometry**. In order to bring this geometry to the screen, numerous attributes and processes are required:

- Position in the **world**
- **View** in the world
- **Projection** of the world
- Lighting
- Shadows
- Textures
- Rasterisation
- Pixel tests
- Blending

These processes are dealt with in the XNA **graphics pipeline**.
In general, the XNA 3D pipeline requires the following for initialisation:

1. World, view, and projection transforms to transform 3D vertices to 2D
2. A set of vertices that contains the geometry to render
3. An effect that sets the render state necessary for drawing the geometry

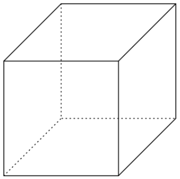
World, View and Projection

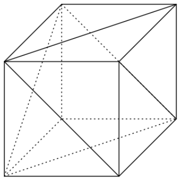
The world, view and projection transforms are required to map a 3D object onto a 2D surface:

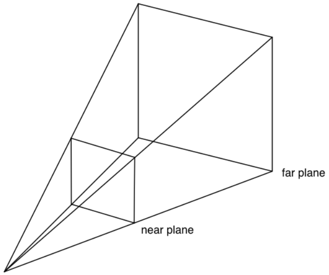
- **World**: describes the position of a particular object in the world. This includes information regarding the geometry's **translation**, **rotation** and **scale**.
- **View**: describes our view within the world. This includes the position of the **camera** and the target the camera points at.
- **Projection**: describes the **viewing frustum**. A viewing frustum is the extend to which we can see along the direction we are looking at.

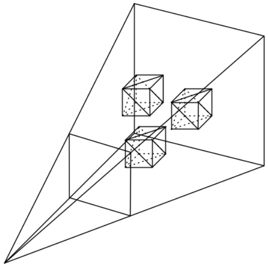
Together, these transforms determine which parts of the object are visible and hence drawn onto the screen. In order to create a realistic depiction (i.e., one that does not look flat), we also make use of **effects**.

What is 3D?









Create View and Projection

To do the drawing of a 3D object: first, create a simple, stationary **camera**.

The concept of a camera may be used to implement the View and Projection:

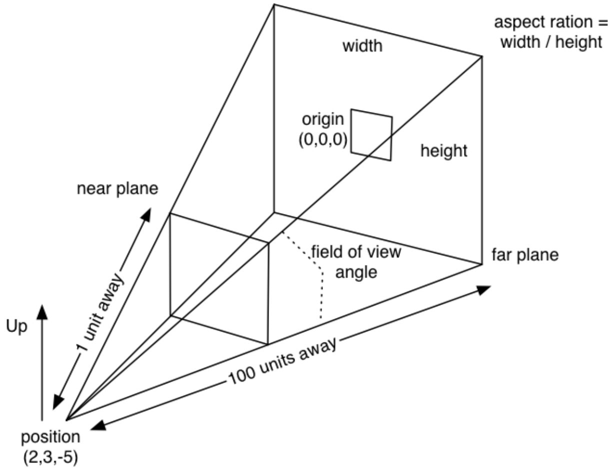
```
Matrix View = Matrix.CreateLookAt(new Vector3(1, 2, 7), Vector3.Zero,
     Vector3.Up)
```

This creates the view. The camera is at position $(1, 2, 7)$ and looks at point $(0, 0, 0)$. We also define which way is up.

```
Matrix Projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.
     PiOver4, GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f));
```

This creates the viewing frustum: the angle specifies the field of view angle, the aspect ration determines the dimensions of the viewing rectangle, the value of 1 is the distance of the near field and 100 is the distance of the far field.

The Viewing Frustum



3D: How to Draw Primitives

There are 4 primitive types in XNA define by the enumeration `PrimitiveType`:

1. `LineList`
2. `LineStrip`
3. `TriangleList`
4. `TriangleStrip`

These types determine how the **vertices** are combined to form a geometry. XNA has several different built-in vertex types:

1. `VertexPositionColor` – holds position and color
2. `VertexPositionColorTexture` – holds position, color and texture
3. `VertexPositionTexture` – holds position and texture
4. `VertexPositionNormalTexture` – holds position, normal and texture

It is also possible to specify custom vertex types.

How to Draw Primitives

There are several ways to draw geometries. Not only can we use different primitive types and vertices, we can also make use of indices (to save memory) and buffers (to improve speed). The `GraphicsDevice` provides the following four methods to draw primitives:

1. `DrawUserPrimitives`
2. `DrawIndexedUserPrimitives` – indexed
3. `DrawPrimitives` – buffered
4. `DrawIndexedPrimitives` – indexed and buffered

Finally, we also need an effect to tell the graphics pipeline how to deal with vertex and pixel shading. For now we will use the built-in effect `BasicEffect`.

In the following examples, we will use the View and Projection defined earlier and for the World we use `Matrix.Identity`.

Drawing Triangles

1. `TriangleList`: The data is ordered as a sequence of triangles; each triangle is described by three new vertices. Backface culling is affected by the current winding-order render state.
2. `TriangleStrip`: The data is ordered as a sequence of triangles; each triangle is described by two new vertices and one vertex from the previous triangle. The backface culling flag is flipped automatically on even-numbered triangles.

What is **backface culling**?

In order to improve speed, triangles that can't be seen are not drawn. This is called culling. By default, triangles are define clockwise in XNA. Counter-clockwise triangles are culled. Culling can be turned off using the `CullMode` of `GraphicsDevice.RasterizerState`.

Drawing Triangles

To draw a triangle, we first, we create some vertices:

```
VertexPositionColor[] vpc;
```

```
vpc = new VertexPositionColor[3];
vpc[0] = new VertexPositionColor(new Vector3( 0,  1, 0); Color.Red);
vpc[1] = new VertexPositionColor(new Vector3( 1, -1, 0); Color.Green);
vpc[2] = new VertexPositionColor(new Vector3(-1, -1, 0); Color.Blue);
```

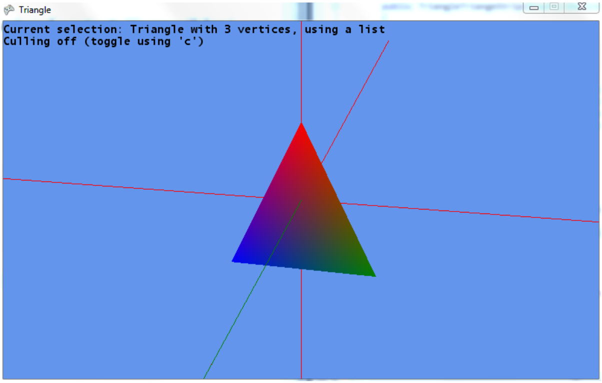
We then apply the effect and draw the triangle as follows:

```
device.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.
     TriangleList, vpc, 0, 1);
```

```
device.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.
     TriangleStrip, vpc, 0, 1);
```

Remember: we can use different methods to use indices and buffers.

Drawing Triangles



```
Current selection: Triangle with 3 vertices, using a list
Culling off (toggle using 'c')
```

World Transforms

The world matrix specified how each individual object is positioned in the world. You can transform the world matrix of each object to induce **rotation**, **translation** or **scale**.

Translation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Scale:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

X-Rotation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\ominus & -\sin\ominus & 0 \\ 0 & \sin\ominus & \cos\ominus & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-Rotation:

$$\begin{bmatrix} \cos\ominus & 0 & \sin\ominus & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\ominus & 0 & \cos\ominus & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z-Rotation:

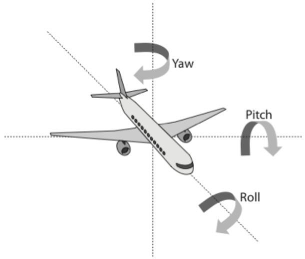
$$\begin{bmatrix} \cos\ominus & -\sin\ominus & 0 & 0 \\ \sin\ominus & \cos\ominus & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

XNA Matrices

XNA provides numerous method and properties (similar to Vector) to obtain specific matrices and to perform operations on these. To obtain the matrices from the previous slide:

- `Matrix.CreateTranslation()`
- `Matrix.CreateScale()`
- `Matrix.CreateRotationX()`
- `Matrix.CreateRotationY()`
- `Matrix.CreateRotationZ()`

Another very useful matrix is `CreateFromYawPitchRoll` which can rotate an object around multiple axes simultaneously.



Matrices & Vectors

Matrices represent linear transformations used to transform vectors. Using a 4×4 matrix allows one to combine all these transformations into a single matrix. Matrices in XNA are **row major**:

$$\begin{bmatrix} R_x & R_y & R_z & R_w \\ U_x & U_y & U_z & U_w \\ -F_x & -F_y & -F_z & -F_w \\ T_x & T_y & T_z & T_w \end{bmatrix}$$

Right (direction)
Up (direction)
-**F**orward = Backward (direction)
Translation (position)

Remember: each object has its own world matrix. The first three rows specify its **local** coordinate system. When you transform this matrix with another matrix (e.g., rotation matrix), all axes will be rotated to reflect the change while maintaining their relationship to one another (usually you don't want to change the orthonormality).

World Transforms

Now, we would like to carry out multiple transformations at once. It is important to note that the order in which individual transformations are applied are important. Rotations, for instance, are always rotations around the origin.

To apply all transforms simultaneously, we can combine them as follows:

```
model.Draw(scale * rotation * translation, view, proj);
```

Now, what would happen in the following cases?

```
model.Draw(scale * translation * rotation, view, proj);
```

```
model.Draw(rotation * translation * scale, view, proj);
```

In general, you will want to apply the transformations in the following order:

Scale $\longrightarrow$ **Rotate** $\longrightarrow$ **Translate**

Outline

What are Models?

Models are really just geometries, usually too complex to specify using primitives. They often contain additional aspects such as colours, textures or even information regarding animations.

Models in XNA are stored as type `Model` and have the following composition:

- `Meshes`: a collection of `ModelMesh` objects, each of which has:
    - `MeshParts`: collection of `ModelMeshPart` objects with:
        - `Effect`, `IndexBuffer`, `VertexBuffer`, `NumVertices`, `PrimitiveCount`
    - `BoundingSphere`: bounding sphere for the mesh
    - `Effects`: collection of effects (one for each mesh part)
    - A `Draw()` method (to draw all mesh parts)
    - `ParentBone`: contains transformation matrix with relative position
- `Bones`: a collection of `ModelBone` objects which have
    - `Parent`, `Children`, `Transform`
- `Root`: the root bone

`Model` also has some methods to efficiently obtain the transformations (bones).

Drawing Models

```
public void DrawModel(Matrix view, Matrix proj)
{
  Matrix[] transforms = new Matrix[model.Bones.Count];
  model.CopyAbsoluteBoneTransformsTo(transforms);

  foreach (ModelMesh mesh in model.Meshes)
  {
    foreach (ModelMeshPart mmp in mesh.MeshParts)
    {
      BasicEffect effect = mmp.Effect as BasicEffect;
      effect.EnableDefaultLighting();
      effect.Projection = proj;
      effect.View = view;
      effect.World = world * mesh.ParentBone.Transform;

      effect.GraphicsDevice.SetVertexBuffer(mmp.VertexBuffer, mmp.
           VertexOffset);
      effect.GraphicsDevice.Indices = mmp.IndexBuffer;

      foreach (EffectPass ep in effect.CurrentTechnique.Passes)
      {
        ep.Apply();
        device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
             mmp.NumVertices, mmp.StartIndex, mmp.PrimitiveCount);
      }
    }
  }
}
```

Note: need to apply effect.

Note: `BasicEffect` only has one pass so third loop not really required.

Calculating Bounding Spheres

Collision detection between objects in the game is essential. We can use bounding spheres to get a rough idea as to which portions of a 3D world a particular model occupies.

```
BoundingSphere boundingSphere;
```

Create an all-ecompassing bounding sphere from all the model's meshes:

```
private void buildBoundingSphere()
{
  Matrix[] transforms = new Matrix[model.Bones.Count];
  model.CopyAbsoluteBoneTransformsTo(transforms);

  boundingSphere = new BoundingSphere(Vector3.Zero, 0);

  foreach (ModelMesh mesh in model.Meshes)
  {
    BoundingSphere transformed = mesh.BoundingSphere.Transform(
         transforms[mesh.ParentBone.Index]);
    boundingSphere = BoundingSphere.CreateMerged(boundingSphere,
         transformed);
  }
}
```

Calculating Bounding Spheres

Remember: It is necessary to scale and translate the bounding sphere in accordance to the model to match the two.

In summary:

- Each mesh of the model has its own bounding sphere
- To get the model's bounding sphere, merge all spheres together
- Need to translate and scale sphere by the mesh's transform
- Use `BoundingSphere.Contains()` and `BoundingSphere.Intersects()`

Q What might be problematic with the use of bounding spheres?

Always bear in mind: accuracy vs efficiency.

Q What is an efficient approach to good collision detection (on a single multi-mesh object)?

Bounding Boxes

A sphere is very useful to approximate the shape of complex objects (e.g., spaceship with wings etc.). However, in some cases, a bounding box is a much better fit. Unfortunately, XNA does not provide a readily available bounding box. There are two solutions to this:

- Create a bounding box from a bounding sphere using `BoundingBox.CreateFromSphere()`
- Create a bounding box from 2 points and try to make it fit the object `BoundingBox box = new BoundingBox(Vector3 min, Vector3 max);`

Using the latter, we can create a bounding box as follows:

```
Vector3 min = new Vector3(0f);
Vector3 max = new Vector3(1f);
from = Vector3.Transform(min, objectWorld);
to = Vector3.Transform(max, objectWorld);
BoundingBox box = new BoundingBox(min, max);
```

Outline

Pathfinding

Amongst the most fundamental game AI is path-finding, an ability central to most interactions of NPCs and gamers. Simple Path-finding AI combined with some scripted behaviours may lead to a sufficiently strong opponent.

First, we need to construct a graph that we can search:

- Nodes represent points in 2D or 3D space
- Nodes have neighbours (adjacent nodes)
- Neighbouring nodes may have different distances
- Nodes (i.e., their locations) can be constructed manually or automatically

The set of nodes that make up the graph are known as the pathfinding network.

We need a way to construct the graph.

- Discretise the continuous space using grids
- Simplest: regular grids (rectangular or hexagonal)
- Rectangular grid: 4- or 8-way connectivity

A*

To obtain the shortest paths of a weighted graph, we can use Dijkstra's algorithm. Dijkstra uses the distance travelled so far and utilises a priority queue to consider nodes in the graph with the smallest distance from the start found so far.

We would expect to perform better if we could also guess how close a node is to the target (for point-to-point distances).

A* uses a **heuristic** to estimate the utility of a node with respect to the target. This allows the algorithm to explore the graph towards the more promising regions. For this to work, we require the following:

- The heuristic must be **admissable**.
- An addmissable heuristic never over-estimates the distance from A to B
- The closer the heuristic to the actual distance, the better A* performs.

Q Why is it important not to overestimate the cost/distance to the target?

A* I

```
public enum Heuristics { STRAIGHT, MANHATTEN };

public static List<N> AStar(N start, N target, Heuristics heuristic)
{
  PQ open = new PQ();
  List<N> closed = new List<N>();
  start.g = 0;
  start.h = GetHValue(heuristic, start, target);
  open.Add(start);

  while (!open.IsEmpty()) {
    currentNode = open.Get();
    closed.Add(currentNode);

    if (currentNode.isEqual(target)) break;

    foreach (E next in currentNode.adj) {
      double currentDistance = next.cost;

      if (!open.Contains(next.node) && !closed.Contains(next.node)) {
        next.node.g = currentDistance + currentNode.g;
        next.node.h = GetHValue(heuristic, next.node, target);
        next.node.parent = currentNode;
        open.Add(next.node);
      }
      else if (currentDistance + currentNode.g < next.node.g) {
        next.node.g = currentDistance + currentNode.g;
        next.node.parent = currentNode;

        if (open.Contains(next.node))
```

A* II

```
          open.Remove(next.node);

        if (closed.Contains(next.node))
          closed.Remove(next.node);

        open.Add(next.node);
      }}}
  return ExtractPath(target);
}
```

To use A*, we need a heuristic. In the simplest case, we can say $h = 0$. A* then behaves identical to Dijsktra. In order to improve the algorithm, we should choose an admissable heuristic as close as possible to the real distance.

Q What other attribute is important for the heuristic to be useful?

Heuristics for rectangular grids:

- 4-way: straight-line / Euclidean distance, Manhattan distance
- 8-way: straight-line distance / Euclidean, Diagonal / Chebychev distance

In some cases we can also use a pre-computed exact heuristic.

Some Facts about A*

Some facts from `theory.stanford.edu/~amitp/GameProgramming/Heuristics.html`:

- If $h(n)$ is 0, only $g(n)$ matters, and A* equals Dijkstra's algorithm.
- If $h(n)$ is never more than the actual distance, then A* is guaranteed to find a shortest path. The lower $h(n)$, the more node A* expands.
- If $h(n)$ is exactly equal to the actual distances, then A* will only follow the best path and never expand anything else.
- If $h(n)$ is sometimes greater than the cost of moving from n to the goal, then A* is not guaranteed to find a shortest path.
- If $h(n)$ is very high relative to $g(n)$, then only $h(n)$ plays a role, and A* turns into Best-First-Search.

Outline

Behaviour Trees

We use the notation and examples from Millington and Funge (ch. 5).

Behaviour trees have become very popular in recent years (starting 2004) as a more flexible, scalable and intuitive way to encode complex NPC behaviours. One of the first popular games to use BTs was Halo 2.

Some of the advantages of BTs are as follows:

- Can incorporate numerous concerns such as path finding and planning
- Modular and scalable
- Easy to develop, even for non-technical developers
- Can use GUIs for easy creation and manipulation of BTs

Tasks can be composed into sub-trees for more complex actions and multiple tasks can define specific behaviours.

A Task

A task is essentially an activity that, given some CPU time, returns a value.

- Simplest case: returns success or failure (boolean)
- Often desirable to return a more complex outcome (enumeration, range)

Task should be broken down into smaller complete (independent) actions.

A basic BT consists of the following 3 elements:

- **Conditions**: test some property of the game
- **Actions**: alter the state of the game; they usually succeed
- **Composites**: collections of child tasks (conditions, actions, composites)

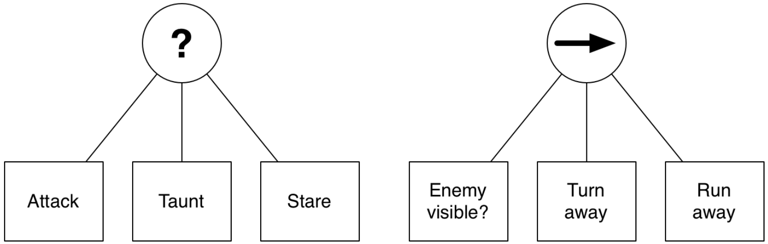
Conditions and actions sit at the leaf nodes of the tree. They are preceded by composites. Actions can be anything, including playing animations etc.

Composite tasks (always consider child behaviours in sequence):

- **Selector**: returns success as soon as one child behaviour succeeds
- **Sequence**: returns success only if all child behaviours succeeds

Selectors and Sequences

Two simple examples of selectors and sequences:



Q What do Selector and Sequence correspond to in term of logical operands?

Order of Actions

We can use the order of actions to imply priority:

- Child actions are always executed in the order they were defined
- Often a fixed order is necessary (some actions are prerequisites of others)
- Sometimes, however, this leads to predictable (and boring) behaviour

Imagine you need to obtains items A, B and C to carry out an action. If the items are independent of one another, one obtains a more diverse behaviour if these items are always collected in a different order.

We want to have **partial ordering**: some strict order mixed with some random order. We thus use two new operators, random Selector and random Sequence.

Note: you can see from this that the design of behaviours does not only have to consider functionality (i.e., getting the job done) but also gameplay experience.

Outline

Billboarding

Billboarding is a technique where 2D textures are drawn onto 3D rectangles (quads) placed in the scene. Usually, the quads are rotated towards the camera such that they will always face the viewer. This allows one to create the illusion of a model albeit at a much reduced cost. A common usage is trees in the background or clouds in the sky. The types of billboard we consider are:

- Cylindrical
- Spherical
- Non-rotating (interleaved)

It is common to use billboards in the distance and replace them with models once the gamer gets sufficiently close. This is a common approach in many "level of detail" systems.

The implementation of billboards is based on chapter 6 in "3D Graphics with XNA Game Studio 4.0" by S. James but has been simplified.

Billboarding

The prerequisite for billboarding are textured quads. We also require a set of positions of where to draw the quads and an effect that

- Rotates the billboards towards the camera
- Deals with transparency
- Is able to ensure proper occlusion

Q What information do we need to perform the rotation?
Q Why would we want the effect to perform the rotation?

A general billboarding system requires the following:

- A quad (using index and vertex buffers)
- Number of billboards
- Size of each billboard
- Position of each billboard
- Texture to be drawn (e.g., tree, cloud)
- An effect

Billboarding

We do the rotation in the effect (**why?**). First, we have to set all the parameters for the effect:

```
void setEffectParameters(Matrix View, Matrix Projection, Vector3 Up,
    Vector3 Right)
{
  effect.Parameters["ParticleTexture"].SetValue(texture);
  effect.Parameters["View"].SetValue(View);
  effect.Parameters["Projection"].SetValue(Projection);
  effect.Parameters["Size"].SetValue(billboardSize / 2f);
  effect.Parameters["Up"].SetValue(Up);
  effect.Parameters["Side"].SetValue(Right);

  effect.CurrentTechnique.Passes[0].Apply();
}
```

We need to supply the up and right vectors of the camera to make sure we can rotate the billboard properly.

The draw method then simply:

- Sets the buffers
- Sets the effect parameters (call to method `setEffectParameters()`)
- Draws the billboard (using built-in draw method)

Billboarding Effect I

```
float4x4 View;
float4x4 Projection;
float2 Size;
float3 Up;
float3 Side;
texture ParticleTexture;

sampler2D texSampler = sampler_state
{
  texture = <ParticleTexture>;
};

struct VertexShaderInput
{
  float4 Position : POSITION0;
  float2 UV : TEXCOORD0;
};

struct VertexShaderOutput
{
  float4 Position : POSITION0;
  float2 UV : TEXCOORD0;
};

VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
{
  VertexShaderOutput output;

  float3 position = input.Position;

```

Billboarding Effect II

```
  // Determine which corner of the rectangle this vertex represents
  float2 offset = float2((input.UV.x - 0.5f) * 2.0f, -(input.UV.y -
       0.5f) * 2.0f);

  // Move the vertex along the camera's 'plane' to its corner
  position += offset.x * Size.x * Side + offset.y * Size.y * Up;

  // Transform the position by view and projection
  output.Position = mul(float4(position, 1), mul(View, Projection));
  output.UV = input.UV;

  return output;
}

float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
{
  float4 color = tex2D(texSampler, input.UV);

  return color;
}

technique Technique1
{
  pass Pass1
  {
    VertexShader = compile vs_2_0 VertexShaderFunction();
    PixelShader = compile ps_2_0 PixelShaderFunction();
  }
}
```

Billboarding

The most interesting part of the effect is the vertex shader.

```
VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
{
  VertexShaderOutput output;
  float3 position = input.Position;

  float2 offset = float2((input.UV.x - 0.5f) * 2.0f, -(input.UV.y -
       0.5f) * 2.0f);
  position += offset.x * Size.x * Side + offset.y * Size.y * Up;

  output.Position = mul(float4(position, 1), mul(View, Projection));
  output.UV = input.UV;

  return output;
}
```

First we get the position of the vertex (POSITION0). We then calculate the offset (remember, all vertices are at the same location when the quad is created), including the camera's local coordinate system. Finally, once we have the position, we transform it via the view and projection.

Billboarding

Almost done. The empty regions of the billboard are visible. We have to:

- Enable *alpha blending* (to get transparency)
- Draw only 'solid' pixels (to get proper occlusion)
  - Discard all pixels with an alpha value less than x
  - We check for this in our pixel shader

Spherical billboarding might be inappropriate (e.g., for trees). It would be better to use **cylindrical** billboarding: rotate around one axis only (up in this case).

We thus add the option to switch between the two. Replace:

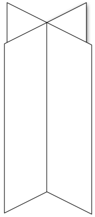
```
effect.Parameters["Up"].SetValue(Up);
```

with

```
effect.Parameters["Up"].SetValue(Mode == BillboardMode.Spherical ? Up
     : Vector3.Up);
```

Non-Rotating Billboarding

We now need to draw to rectangles at 90 degrees to one another. This time we specify the exact positions in 3D space since we require no further changes to that.



We can use much of the code we developed earlier and change:

- We need to draw two quads instead of one
- There is no choice between spherical and cylindrical rotations
- We thus don't need the up and right / side vectors
- The effect can be simplified (no rotation required)
- We should turn off culling to make sure the trees are visible from all sides