public void updateMovableObject() {
    Vector3 startLocation, stopLocation;
    startLocation = stopLocation = Translation;
    // translate to origin
    Orientation *= Matrix.CreateTranslation(-1 * Translation);
    Orientation *= Matrix.CreateRotationY(yaw);  // rotations
    Orientation *= Matrix.CreateRotationX(pitch);
    Orientation *= Matrix.CreateRotationZ(roll);
    stopLocation += ((step * stepSize) * Forward);  // go forward
    if (model.IsCollidable && collision(stopLocation)) {
        Orientation *= Matrix.CreateTranslation(startLocation);
        return;  // don't move
    }  // no collision test if move on terrain
    if (stage.withinRange(this.Name, stopLocation))  // go forward
        Orientation *= Matrix.CreateTranslation(stopLocation);
    else // off terrain, reset location, don't move
        Orientation *= Matrix.CreateTranslation(startLocation);
}
Pattern (Path) movement

NPC move in "pre-set" animation sequences (graphs, paths)

Path based used a collection of nodes: translation or orientation interpolate between nodes, or orient to next node, step to goal
Path can be generated by "walking and clicking tool"

Example path using a collection of transformations and speed.

PathNode { Vector3 rotationAxis; float radians, speed; } List <PathNode> path;

    each iteration of render loop increment's character's next step indexes into pattern.

Control sequence can be statically or dynamically determined from grammars...

0 initialize
1 y += -90°, step
2 step
3 y += -90°, step
4 y += 90°, step
5 step
6 step
Navigation graphs

Information about the local area can be stored at locations in the scene: navigation nodes, way points, path (nodes), ...

"cost" to traverse terrain
objects in local area

Nodes are in a connected graph.

Nodes provide NPC a limited visibility detection – can detect n nodes from current node.

Nodes limit NPC navigation along "correct" paths.

Movable objects translation is interpolated along, or constrained by, path.

Nodes can be placed by designer, equally spaced from a terrain grid, or by space partitioning (Quad or Oct Tree – later topic).
AGMGSK Path following

A path is a list of Navigation Nodes (NavNodes)
   Each NavNode has a position (aka Translation) // also other fields...

In AGMGSK NPAgent follows its path using its NavNode nextNode
   NPAgent turnToFace(nextNode)
   NPAgent moves stepSize (10) on each update.
   when it is within snapDistance (20) of its nextNode it "snaps" to NextNode // property handles end of list
   nextNode is updated
end of list behavior determined by type: single, loop, reverse

Single path sets done true when at last node in path, update shouldn't move

Loop path sets nextNode to path's first node when at the last node in path.

Reverse path, when at last node in list, reverse's path list and sets nextNode to second node in path.
public class NavNode : IComparable<NavNode> {
    public enum NavNodeEnum {
        VERTEX, WAYPOINT, PATH, OPEN, CLOSED
    }
    private double distance;  // for use with A* path finding.
    private Vector3 translation;
    private NavNodeEnum navigatable;
    private Vector3 nodeColor;  // determined by NavNodeEnum

    public NavNode Enum Navigatable {
        get { return navigatable; }
        set { navigatable = value;
            switch (navigatable) {
                case NavNodeEnum.VERTEX :
                    nodeColor = Color.Black.ToVector3(); break;
                case NavNodeEnum.WAYPOINT :
                    nodeColor = Color.Yellow.ToVector3(); break;
                case NavNodeEnum.PATH     :
                    nodeColor = Color.White.ToVector3(); break;
                case NavNodeEnum.OPEN   :
                    nodeColor = Color.BLUE.ToVector3(); break;
                case NavNodeEnum.CLOSED  :
                    nodeColor = Color.RED.ToVector3(); break;
            }
        }
    }
    ...
}
class Path's Draw(...) { // method
    if (! drawNodes) return;
    Matrix[] modelTransforms =
        new Matrix[stage.WayPoint3D.Bones.Count];
    foreach(NavNode navNode in node) { // List<NavNode> node
        // if (node.Navigatable == NavNode.NavNodeEnum.VERTEX)
        //    continue; // don't draw
        // do model's meshes code
        // now set lights and blending to "color" navNode
        effect.DirectionalLight0.DiffuseColor = navNode.NodeColor;
        effect.AmbientLightColor = navNode.NodeColor;
        effect.DirectionalLight0.Direction = stage.LightDirection;
        effect.DirectionalLight0.Enabled = true;
        // set effect.View, effect.World, effect.Projectiion
        stage.setBlendingState(true);
        mesh.Draw();
        stage.setBlendingState(false); // reset blend state
    }
Kinetics

Physical modeling: velocity, acceleration, inertia, gravity, friction,... requires continuous time to be "packetized".

Time deltas (dt) are timer ticks or elapsed time per render frame

Motion -- object might have:
- position (@) center of bounding sphere or box
- direction vector (DV) where length is speed
  - for each dt @ += DV
- normalized DV (Forward) for collision reflection, viewing

Error (e) dt ± e collision point ± e collision time ± e
determine acceptable range of accuracy (speed / time unit)
accuracy Vs performance tradeoff
Error in Movement

No error – step size (distance / tick) is multiple of distance between objects.

\[ d = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2 + (z_1-z_2)^2} \]

Error – step size not multiple of distance between objects. At time \( t_{i+2} \) collision is detected late (\( \varepsilon \)).
**Collision Detection**: computationally intense

2 or more objects are moving - do they collide?  
\[ O(n^2) \quad n = \text{faces} / \text{object} \]

How often is collision tested (collision culling)  
Test when predictive collision is possible

Multi-phase approach (simple example)

Cull all objects pairs that exceed collision thresholds (distance from centers)

Test spherical bounding volumes

Test axis - aligned box bounding volumes

Test axis - oriented box bounding volumes

Test each surface for collision

Spherical test  \((r_1 + r_2 \geq \text{distance between centers})\)

often sufficient.
Multiple Bounding Spheres

Multiple bounding spheres can be used to cover non-spherical shapes with better approximation.

Collision test is against a collection of object's bounding spheres.
Colliding moving objects (Sphere == simplicity)

distance between spheres <= sum radii
axis aligned BB tests coordinate values
axis oriented BB tests vector – plane intersection wrt dt

Reflected direction by collision between object and stationary surface (plane) is determined by reversing the direction (sign) of DV and a $\pi$ radian, 180°, rotation of colliding vector with the surface normal.

Error, $\varepsilon$, @ when collision detected is not on the plane.

Collision between two moving objects is a $\pi$ radian rotation about the vector between their centers (w/ equal inertia) at their centers.
Collision with plane’s axis aligned bounding box

Collision with plane’s axis oriented bounding box.

Error possible both in direction and distance.

Collision with plane’s bounding sphere

Error with distance test for collision rather than DV axis intersection test.
MonoGames Collisions

Models have a Meshes property that get a ModelMeshCollection of ModelMesh objects.

ModelMeshes have a BoundingSphere property that gets a BoundingSphere structure

BoundingSphere are spheres that can be used for intersection tests.

See MonoGames Documentation Math Overview | Testing for Intersection and Containment of Geometry

float radius
vector3 center

bool intersects(BoundingSphere)
bool intersects(BoundingBox) // BoundingBox MonoGames struct
bool intersects(Plane) // Plane MonoGames struct
bool intersects(Ray) // Ray MonoGames struct

BoundingBox is axis aligned.
Collisions of models

// Do 2 Models intersect?
void CheckForCollisions(Model m1, Model m2) {
    for (int i = 0; i < m1.Meshes.Count; i++) {
        // Do the bounding boxes of the two models intersect?
        BoundingSphere m1BoundingSphere = m1.Meshes[i].BoundingSphere;
        m1BoundingSphere.Center += m1.position;
        for (int j = 0; j < m2.Meshes.Count; j++) {
            BoundingSphere m2BoundingSphere = m2.Meshes[j].BoundingSphere;
            m2BoundingSphere.Center += m2.position;
            if (m1BoundingSphere.Intersects(m2BoundingSphere)) {
                // deal with collision
            }
        }
    }
}

BoundingSpheres can be created from vertex collections.

public static BoundingSphere CreateFromPoints (IEnumerable<Vector3> points

Selection w/ Rays

A Ray structure represents a vector (position and unit vector for direction with intersect methods)

```csharp
public Ray ( Vector3 position, Vector3 direction )

    bool intersects(BoundingSphere)
    bool intersects(BoundingBox)
    bool intersects(Plane)
```

In a 3D scene an avatar might have a selection object: magic wand, weapon, telescope ...

Assume that the selection object (selectOM) has an orientation matrix. Then its intersection ray is:

```csharp
Ray selectionRay =
    new Ray(selectOM.Translations, selectOM.Forward);
```

Select by testing for ray intersection with BoundingSpheres of all SelectableObjects.
// Windows (only) systems can use MouseState
MouseState mouseState = Mouse.GetState();
int mouseX = mouseState.X;
int mouseY = mouseState.Y;

// Get mouse position on near and far clipping planes
// projection frustrum is "normalized"
Vector3 near = new Vector3((float)mouseX, (float)mouseY, 0f);
Vector3 far = new Vector3((float)mouseX, (float)mouseY, 1f);

// Use Viewport.Unproject to get 3D position of mouse
Matrix world = Matrix.CreateTranslation(0, 0, 0);
Vector3 near =
    graphics.GraphicsDevice.Viewport.Unproject(near, proj, view, world);
Vector3 far =
    graphics.GraphicsDevice.Viewport.Unproject(far, proj, view, world);

Test for Ray intersections with all SelectableObjects.
Obstacle Avoidance

Obstacles avoidance requires "see ahead" information.

Let NPCs have "sensors" ahead of (around?) them.

Sensors detect potential collisions with obstacles. Detection determines avoidance steering.

In the lower diagram, avatar can see its target / goal ("green" object).

Ray intersection can be used for line-of-site detection of targets.

The avatar can't fit between the "red" obstacles.

Determine the "best" path to target using "collision avoidance" and line-of-site?
Algorithm for this movement behavior?
What is the decision here?

12 steps
7 orientations

11 steps
6 orientations
State based avoidance

npAgent moves and evaluates right and left sensors and states

- rightSensor \{ T \parallel F \}
- rightWall \{ T \parallel F \}
- goalPathLost \{ T \parallel F \}

- leftSensor \{ T \parallel F \}
- leftWall \{ T \parallel F \}
- other states … ???

Given states what actions? backup, turnRight, turnLeft, …
Start to seek target (goal), move towards target hit wall, follow until no wall, lost goal, move past corner, re-acquire goal
Trapped, backup?

Goal, oscillates between rightWall (turn left) || leftWall (turn right) and forward. Can’t get out of corner.

Possible additional state: action

leftHit && rightHit:
backup( nSteps)

Still needs “tuning” for nSteps

FSM complexity grows ....
Detection, path finding

Assume navGraph, is a graph of NavNodes used for moving Avatars and NPAvatars in a scene. Avatars regularly "land" on NavNodes.

```csharp
NavNode () { // or NavGraphNode : NavNode ...
  Vector3 location;
  int distance, distanceToSource, distanceToGoal;
  List<NavNode> adjacent; // regular has 8 possible
  NavNode pathPredecessor; }
```

NPCs can then detect, chase, or evade other Avatar's by searching their adjacent area w/in their visibility (detection).

For example vertical and horizontal moves to adjacent NavNodes can have a cost, or score, of 10 and diagonal of 14 to control for distance.

A* shortest path can be used for detection or path finding:

```csharp
NavNodes source, goal // NPC's path start and stop locations
cost = cost + heuristic // cost is usually distance
distance = distanceToSource + "estimated" distanceToGoal
```
A* algorithm

Collection open, closed, path  // empty or null
  // open set ordered on minimum cost
open += source node  // cost = 0, null pathPredecessor
bool done = false;
while (open ! empty)
cur = open's lowest cost node
  if cur == goal then done = true  // path found
  else
closed += cur  // mark node as processed
  foreach of cur's adjacent nodes
    if (node !in open && !in closed)
      cost = known + estimated  // to source + to goal
      open += node  // mark node as partially processed
      node.predecessor = cur  // back reference
if (done)  // path exists, build path
  Collection path += traverse back to source w/ node predecessors
else if (open == empty) no path exists

A heuristic (distance to goal) can be used to estimate node's cost.
cost = distanceFromSource + distanceToGoal  // Euclidian distance
or difficulty to navigate terrain – roughness, incline, ...
A* example (compare to bfs, dijkstra)
A* with C# collections

C# Collection List<T> class has methods for:

```csharp
    public void Sort()  // ascending, PQ == open.Sort()
    public bool Contains(T item)  // membership test
```

Consider class for a graph for path finding, and modify and/or extend NavNode to include aspects of a NavGraphNode...

```csharp
class NavGraphNode : NavNode
    private int cost;  // function of distance
    private double distanceToSource, distanceToGoal;
    private List<NavGraphNode> adjacent;
    private NavGraphNode pathPredecessor;
    public int CompareTo(NavGraphNode n)  // on "cost"
...
```

The navigation graph is probably a `Dictionary<K,V>`

```csharp
    Dictionary <String, NavGraphNode> graph
```

Dictionary properties and methods demonstrated in `GraphTest.cs` example
NavGraph<K, NavGraphNode>

class NavGraph
Dictionary <String, NavGraphNode> graph  // Key "x:z"
List<NavGraphNode> open, closed, path;
List<NavGraphNode> aStarPath( ... )
...

// Example key for graph based on (x,z)
private String skey(int x, int z) {
    return String.Format("{0}::{1}", x, z); }

// Use index property to access graph as array
// see also Dictionary<K,V> Item property
public NavGraphNode this [x, z] {
    get { return graph[(skey(x, z))]; } 
    set { graph[skey(x, z)] = value; }
}

// Example of foreach usage

// KeyValuePair<K,V> Dictionary's internal type
// *.Value Dictionary's property for stored values
private void setAll (NavNode.NavNodeEnum nodeType) {
    foreach (KeyValuePair<String, NavGraphNode> item
        in graph)
        item.Value.Navigatable = nodeType;  
}
Regular NavGraphNodes

NavNodes are at regular spacing (1/vertex?)
Navigate w/ Regular NavGraphNodes

NPC movement is constrained to orient towards an adjacent "target" NavNode.

Each NavNode has adjacency matrix (8 possible adjacent nodes)

NPC moves towards target NavNode

Within "snap-to" distance NPC is set at target NavNode

Target NavNode becomes current NavNode, NPC orient's towards next "target" NavNode.

+ Constant space partitioning (no "scale" boundaries)
+ Regular NavNodes provide NPC with "constant" rotations.

– Too many regular NavNodes required in a sparse object world.

Use widely spaced NavNodes + NavNodes that surround "collide with" scene's objects
Regular NavGraphNodes with Wall

Terrain is 64 by 64 vertices with spacing = 150, use all valid WayPoints
3,463 nodes  15% sparse  adjacent nodes 27,704  8 / node

yellow is WayPoint
red is closed set
blue is open set
white is path

A* path to goal
A* return path
Regular NavGraph with spacing

203 nodes, 95.04% sparse adjacent nodes 1,336 / node (xDim = 64, halt = 4)
Sparse world w/ NavGraphNodes
"Stamped" regular, sparse navGraph

If there is a non-walkable vertex within "stamp" (halt x halt) graph has navigation nodes for all walkable vertices
else graph has 9 or 5 navigation nodes.

Need to merge close stamp edge nodes?

1,133 nodes, 72.34 % sparse (xDim = 64, halt = 16)
Quad Tree

BSP – binary space partitioning algorithms create trees of space.
   view culling – viewable or non-viewable space
   navigation culling – walkable and non-walkable space

Quad Trees can partition 2D space – or terrain maps
   http://en.wikipedia.org/wiki/Quadtree

Oct Trees partition 3D space.

Navigation w/ QuadTree navNodes at vertices not w/in boundingSphere of any stationary object.

Have “regular nodes” to “sparse level” further partition w/o collisions

```java
createNavNodes(x1, y1, x2, y2)
   set navNodes(...) // no duplicate navNodes
   if ( partitionable(...) && quadHasModel(...) )
      createNavNode(...) // upper right quadrant
      createNavNode(...) // lower right quadrant
      createNavNode(...) // upper left quadrant
      createNavNode(...) // lower left quadrant
```
Quad Tree NavGraphNodes
Quad Tree NavGraphNodes with Wall

953 nodes  77 % sparse
adjacent nodes 12,582  13.2 / node
halts w/o objects at 32 vertices
+
fewer nodes and adjacent nodes
?
adjacent node connection across
partitions of different size
?
path not as optimal
Adjacent Quad NavGraphNodes

In a sparse collection of Quad NavGraphNodes what are a node's adjacent nodes and how are they linked for navigation?

Set and update adjacent nodes as tree is built.
Set on recursive descent, link on ascent?

Set update adjacent nodes after tree is built.
Connect all nodes w/in the recursive level's
\[ \sqrt{2 \times (\text{levelDistance}/2)^2} \]
store values in descent or other suitable max distance to adjacent nodes

For example, when the halt w/o any enclosed objects condition is met.
create 9 NavGraphNodes (return node if it exists) and link adjacents
    current position
    8 surrounding positions

adjacent nodes are linked bi-directionally

Could create 5 NavGraphNodes and link ….
Overlapping adjacency links

NavNodes = 1166 \approx 72\% \text{ sparse}
Adjacent edges = 8577
\approx 7.36 / \text{ node}
Maximum adjacency size = 18

For each nN in graph
for each nA in graph
    if (nN != nA &&
        nN.distance(nA) \leq \sqrt{2*minOffset^2} + 1)
        nN.adjacency += nA
Chase / evade

C, a "predator" chases a non-evading, E, "prey"

Vector3 At is the object's "looking at" -- MonoGames Forward

Vector3 E, C translations to objects

C Orients towards E
Axis to rotate = At.Cross(E-C)
Angle to rotate = cosine (At.Dot (E – C))

Cᵢ position after move
Eᵢ moves

Depending on relative speeds, C can "over shoot" and correct to capture E.

Interception prediction can reduce over shoot.

C moves to where E's next move would be.
C knows, or, estimates E speed and direction.

mid course corrections …
Boids – *flocking*

C. Reynolds’s flocking algorithm’s 3 rules:

Cohesion – boid’s force toward the average position of its neighbors

Separation – boid’s force to avoid hitting neighbors

Alignment – boid aligns to average alignment of its neighbors

Boids (NPCs) have a unit of visibility: arc radius, \( r \), and an angle \( \theta \)

- small \( r \) causes flocks to break into smaller flocks

- \( \theta \) of 270° or 4.71 radians

- good initial angle – "blind spot" behind boid is 90°

- flocks w/ leaders – use leader's position and alignment.
Steering

Average flock position = \( \sum \) boid positions / number of boids

boid’s movement \( \Leftarrow \) flock position force + separation force

force towards flock position is weighted by cohesion
separation force applied to any boid inside separation arc.

before movement boid aligns

flock alignment = \( \sum \) boid Forward.Normalize() / number of boids

boid's rotates towards flock’s alignment (incremental alignment)
about axis \( \Leftarrow \) Cross (boid.Forward, flock alignment)
Force coordination

Force rules can have side effects – may need to "tune" to application

eg – cohesion dominates, flock may not be able to avoid obstacles

Separation force should be weak at larger separation distances
stronger as separation diminishes – an inverse function (not discrete)

\[
\text{separationForce} = \frac{\text{avoidanceForce}}{\text{separationDistance}}
\]

Alignment change proportional to boid's alignment angle and flock'
alignment – only so much rotational change allowed on any adjustment.

Leader based "pack" behavior. Pack "flocks" around leader's alignment
and position.

Leaders can be over-weighted members of flock.

Spreadsheets can be useful in designing expressions for separation and cohesion
Combining Steering Forces

Often several steering forces need to be combined:
- obstacle avoidance, path finding – goal orientation, flocking, ...

Weighted Sum

The resulting directional vector = $\sum (\text{force}_i \times \text{weight}_i)$
vector is then truncated (scaled) to maximum value
- calculates all forces every evaluation step
- forces often conflict
- weight determination

Prioritized Weighted Sum

Compute weighted sum in prioritized order
When sum $> \text{maximum value}$ stop calculation, truncated to max

Prioritized probabilistic evaluation

Each force has a probability of being evaluated on a step and a priority for evaluation.
First force evaluated at or above probability is used.
A dog pack distribution

Packing determines vector to rotate towards on next step from 3 force vectors

\[ V_r = \text{normalize}(V_s + V_a + V_c) \]

Angle of each rotation is small
say $1^\circ$ or 0.017 radians

average dog bounding sphere is 200
**pack = 33%, leader weight = 3**

Leader's weight \{1, 2, 3\} is applied only to the separation force. More weight, more separation.
pack = 99%, leader weight = 3

Flocking simulation
'S' flock prob. {0, 0.33, 0.66, 0.99} 'W' leader's weight {1x, 2x, 3x} 'L' switch leader
leader is Player.0 leader's weight 3 probability = 0.99
pack = 99\%, leader weight = 3

Chaser's alignment is looking at the orange corner cube.

Most dogs are between Chaser and the orange cube.
Flocking is an example of emergent behavior. Other examples are fractals, strange attractors; and cellular Automata (CA).

CA consists of a grid of cells with a set of at most 8 neighbors and a state. Cell state is determined each generation by examining its neighbors. Artificial Life CA has a state of alive or dead. A cell is either born, survives, or dies based on the alive state of its neighbors.

Notationally a cell has a Born_B/Survive_S, Bx/Sy ratio with a B3/S23 ratio a cell is born if it is dead and has 3 alive neighbors, survives if 2 or 3 alive neighbors.

Conway’s Game of Life video simulation B3/S23 animation
Cave generation

Apply a B678/S345678 ratio to a random initial grid with a fixed live border and 50% cells initially alive.

Live cells are walls, and the dead cells as open space.

B678/S345678 animation

Ratio stabilizes quickly ($\approx 15$ generations)
Caves might not be connected.

Lower initially alive cells caves disconnected.
Doesn’t scale well with higher resolution grids (example is $50 \times 50$).

start w/ low resolution, than expand cells. Each cell $\rightarrow$ 4 cells …

Need to
smooth walls of cell…
place floor and roof
Design exercise

How could a series, say N (say 5) cave/grids be used to with AGMGSK to make a cave system with N levels?

How to represent cave_n’s floor and ceiling?

How can Agent walk on cave_i’s floor?

How can Agent move from cave_i to cave_{i+1}?