System Architecture

Code Authoring

This applies to all coders.

Everyone uses MTASc and Swfmill. Anyone who wants to use Flash IDE does so at his/her own peril.
The target Flash player version is 9. This is to be done by compiling .as files as Flash 8 compatible, since the resulting .swf files will run in Flash player 9.
Font concerns:

Coders have no font preferences. Since the arcade machine is a Windows computer, fonts should probably limited to those found in Windows by default or should be provided with the game, if possible.

Source Control:

- There is currently no software source control system in place. Ryan may try to obtain access to an SVN repository that could be used at a later date.
- All source code work is to be submitted to the /work01/workspace/Courses/2006-2007/AC753_Gerstmann_Spring_2007/code directory of the ACCAD ftp server.
- When someone downloads a file from the ftp server with intent to edit it, they must append their first name to the file, ex: “file.as.firstname” or “main.as.chad”.
- When someone submits changes to a file on the server, they must remove their .firstname from the file to indicate that it is no longer being worked on.
- Anyone working on a file should try to submit changes to the file before leaving to do other time consuming activities like sleeping. If the file is not in a tested and usable state by such a time, the file should be submitted to the unstable branch of the code located in */code/unstable.
- If someone wishes to modify a file that is already being modified, they must wait until the other person's version of the file has been submitted and then merge in any changes.

Game Structure

In words:

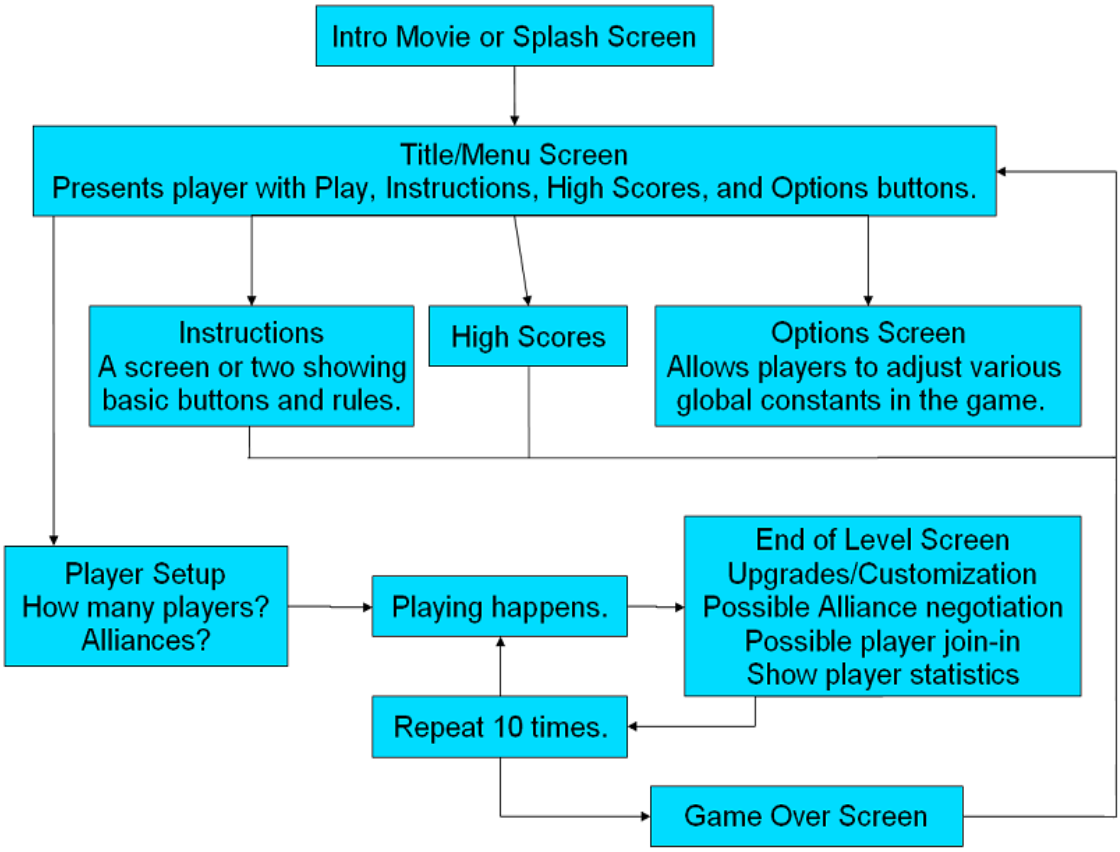
An intro movie will be looped until a player presses a button which brings them to the title screen.

The title screen will have instructions, high score, game, and option screen in the menu.

Upon selecting the game, players customize general play options such as the number of players.

Play commences, and each round consists of a romp in the asteroid field followed by the upgrades screen.

After 10 rounds are up, the game over screen is displayed and the game returns to the title screen.



Class Definitions

The following class definitions specify the basic gameplay engine. At this stage, we seek only to implement spaceflight mechanics: user interface objects for ship customization, high score management, and title screens are omitted. We consider those to be simpler tasks, which can be wrapped around this engine in a future revision. The class definition starts from "bottom up", from simplest objects to most complicated. We focus on architecture and how objects interact via their publicly visible methods, but a few implementation details (private members) are supplied as needed.

Reticle

The atomic UI widget for ship navigation is the `Reticle` class. The publicly visible methods of the `Reticle` class are given below.

```
class Reticle
  {
  public var rx:Number;
  public var ry:Number;

  public function Reticle() : Void;

  public function stickLeftRight (state:Number);
  public function stickUpDown(state:Number);
  public function updatePosition();
  }
```

The member variables `rx` and `ry` are the position of the `Reticle`, in game units. The state of the `Reticle` is modified chiefly through the `stickLeftRight` and `stickUpDown` methods. The `stickLeftRight` method takes a {-1,0,1}:`Number` argument which inform the `Reticle` about the horizontal state of the player joystick: 0 indicates neutral, while +/- 1 indicates the joystick is right/left of center. The other joystick axis has similar semantics. The `updatePosition` method is responsible for updating the `Reticle` position on every frame, using its current knowledge of the joystick state (stored privately). Additional private variables will control the velocity and acceleration state of the `Reticle`, which has yet to be finalized but is hidden from public view.

Particle

The `Particle` class is a data-only definition which specifies position (`rx` & `ry`) , velocity (`vx` & `vy`) and `mass.` All other game entities will be either `Particles` or classes which `extend Particle.` The intent of this class is to encourage regularity among how the (more complicated) offspring actors maintain their game state.

```
class Particle
  {
  public var rx:Number;
  public var ry:Number;

  public var vx:Number;
  public var vy:Number;

  public var mass:Number;
  }
```

SpinningParticle

The `SpinningParticle` is a kind of `Particle` with more variables for representing angular state. The variables `phi` (angular position), `omega` (angular velocity), `moment` (moment of inertia) are the rotational analogs of position, speed and mass. It is our hoped that our physics model can ultimately incorporate angular momentum exchanges in addition to point particle physics. At the very least, we can implement spinning asteroids and spice for "eye candy".

```
class SpinningParticle extends Particle
  {
  public var phi:Number;
  public var omega:Number;
  public var moment:Number;

  SpinningParticle(suppliedID:Number);
  private var avatar:MovieClip;
  private var ID:Number;
  public function expire():Void;
  }
```

Some additional member variables, which are closely tied to the architecture of the Flash/Actionscript virtual machine, appear here for the first time. `SpinningParticle`s are *always* renderable objects. They have an associated `avatar` (display sprite). Like all renderable objects in Flash, it is of type `MovieClip.`

When instantiated by the `new` operator, it is provided a unique integer (`suppliedID`) by the instantiator, through its constructor. It is the responsibility of the `SpinningParticle` to create its `avatar` at instantiation-time, we suggest that it calls `avatar=_root.createEmptyMovieClip(""+suppliedID, suppliedID).` The `expire()` method is analogous to the delete operator – when it is called, the `SpinningParticle` should remove its display sprite by calling `_root.removeMovieClip(""+id);` Practical example: when `Spice` is collected by a player and returned successfully to `HomeBase,` it should be removed from play and disappear, by calling `Spice.expire().`

Tether

The `Tether` is a lasso-like tool that player ships will use to tow objects around, attach to other players, or find other creative uses for. As the "April 3rd Tether Demonstration" depicted, the `Tether` is modeled as a collection of `Particle` "`nodes`" which are stored in an `Array`.

```
class Tether
  {
  private var nodes:Array;
  private var N:Number;

  private var owner:SpinningParticle;
  private var target:SpinningParticle;

  public function setTarget(newTarget:SpinningParticle);
  public function updateVelocity();
  public function updatePosition();


  Tether(suppliedID:Number, suppliedOwner:Ship);
  private var avatar:MovieClip;
  private var ID:Number;
  public function expire():Void;
  }
```

The `updateVelocity` method is responsible for updating the velocities (momenta) of each `node`, using the basic forces of dynamic mass/spring systems. The `owner` and `target` are references for applying (Neumann) boundary conditions – the endpoints of the tether are fixed at the owning ship and target cargo. The `target` for the `Tether` tether (i.e. what object the "other" end is attached to) is established by calling `setTarget`. Supply an argument of `null` to indicate the Tether should release its current `target`.

The `updateVelocity` method is also responsible for applying tension forces to the `owner` and `target` objects (by changing *their* velocities). The `updatePosition` method add-accumulates the current velocity (`vx` & `vy`) into the position (`rx` & `ry`). A requirement of "synchronous" updates will be enforced. Per frame, every game object first executes `updateVelocity` method. Then every object executes its `updatePosition` method. For a given frame, no `updatePosition` method may be called until all `updateVelocity` methods have been called.

The `Reticle` object is the exception to the rule – it is not a true particle but rather a display widget, which does not physically interact with other objects. Its `updatePosition()`method may be called at any point during the frame.

`Tether` also contains all the member fields required for rendering and clearing from the screen (via the `avatar, ID` and `expire` methods). The "minimal effort" artwork for `Tether`

could be a "connect the dots" routine, or use the nodes as control points for a spline for a smoother curve. Its constructor also requires an "owning" `Ship` as input – there is a 1:1 correspondence between `Tethers` and `Ships.`

Ship

The `Ship` class represents player characters (and possibly non-playable computer opponents in future refinements of the game). As a `SpinningParticle`, it inherits the members required for rendering and removing itself from the display.

```
class Ship extends SpinningParticle
  {
  private var tether:Tether;
  private var reticle:Reticle;
  private var attractTarget:SpinningParticle;
  private var repelTarget:SpinningParticle;

  Ship(suppliedID:Number);
  expire();

  public function updateVelocity();
  public function updatePosition();
  public function setAttract(newTarget:SpinningParticle);
  public function setRepel(newTarget:SpinningParticle)
  }
```

The `Ship` encapsulates a `Tether` and a `Reticle` as member variables, because there is a one to one correspondences between all of these objects (each player has a one ship, one reticle and one tether). A `Ship` also has a settor methods for defining the targets of the gravity beam (via `setAttract`) and repulsor beam (via `setRepel`). These arguments can be `null` indicating that the beam has been turned off.

It is our vision that all three of the propulsion methods (gravity beam, repulsor beam, tether) can be simultaneously active. An idle ship can "Reticle over" object A, activate the gravity beam by depressing a button, then "Reticle over" object B and activate the repulsor beam. The gravity force between the `Ship` and object A is sustained as long as the gravity beam button is depressed. A player does not *have* to use all these devices at once. But it is intended that a precocious player is rewarded for quick reflexes and the ability to multitask, by providing additional degrees of freedom to simultaneously control the ship.

The `updateVelocity()` method is responsible for (i) calling the `updateVelocity()`method of the `Tether` (ii) computing the forces of the gravity/repulsor beams (if they are turned on) and accumulating into `vx` & `vy` (iii) applying appropriate forces to the `attractTarget` and `repelTarget,` by summing into their `vx` & `vy` members.

The `updatePosition()` method is responsible for (i) calling the `updatePosition()` method of the `Tether` (ii) updating the `Ships` position (`rx` & `ry`) using the current velocity contents.

Additional Actors

We now introduce additional world objects: the `Spice` which we so desperately crave, the `HomeBase` where we stash our collected booty, and the `Asteroids` (with which we have a very complicated relationship). We love `Asteroids` because they are our primary means of locomotion via the gravity and repulsor beams, but we hate them because they always seem to be in the way, and smashing us.

```
class Spice extends SpinningParticle
  {
  Spice(suppliedID:Number);
  expire();
  public function updatePosition();
  }

class Asteroid extends SpinningParticle
  {
  Asteroid(suppliedID:Number);
  expire();
  public function updatePosition();
  }

class HomeBase extends SpinningParticle
  {
  HomeBase(suppliedID:Number);
  expire();
  }
```

None of these classes have `updateVelocity()`methods – any changes to their velocity have already been imparted by the `Ship` gravity/repel forces and `Tether` tensions, during their `updateVelocity()` calls. Thus, `Spice` and `Asteroid` update their position only. `HomeBase` does not even do that, it is completely stationary. It is still a `SpinningParticle`, however, so that it has a screen renderable `avatar` and a well defined position and angular orientation.

For the first skeleton implementation, these classes are mainly stubs for later functionality (for example, the `HomeBase` class is a logical place to store scoring information, and the `Asteroid` could be smashable apart into `Spice` / other `Asteroids`, etc).

Environment

The Environment class is a global storage container which instantiates and maintains references to all gameplay objects, through arrays of various object types (`Ships, Spices, Asteroids, HomeBases`). It maintains the number of these objects which are active, in member variables of type `Number.`

```
class Environment
  {
  public function Environment()
  private var ShipArray:Array;
  private var N_Ships:Number;

  private var SpiceArray:Array;
  private var N_Spices:Number;

  private var AsteroidLoop:Array;
  private var N_Asteroids:Number;

  private var HomeBaseArray:Array;
  private var N_HomeBases:Number;

  private var next_unique_id:Number;

  public function onEnterFrame();
  public function onKeyUp ();
  public function onKeyDown ();
  }
```

The `Environment` class registers itself to receive key events during its constructor. Some of the messages dispatched to the `onKeyUp` method will be joystick events. In that case, the `Environment` object knows which players joystick was perturbed and in what direction. It should call the appropriate settor method of the `Ships reticle` member. For example:

```
case (Key.UP) :
  ShipArray[0].reticle.stickUpDown (0);
break;
// The up switch of the joystick was just released – stop
// moving the reticle cursor up/down.
```

Some of the messages dispatched to the `onKeyUp` method will be action-button release events. In that case, the Environment object should turn off the gravity beam (or repulsor beam, or `tether`, depending upon which button is pressed).

```
case (Key.Space /* or Key.Shift or Key.Control */):
    ShipArray[0].setAttract(null);
    // or ShipArray[0].setRepel (null);
    // or ShipArray[0].tether.setTarget(null);
break;
```

Similar semantics are applied for `onKeyDown` joystick events. However, action-button downpress events require more effort to handle. The user desires to activate one of their tools on the object currently below their `reticle.` The `Environment` must search through its lists of objects and determine which object, A, is nearest the `reticle` position and then call the `tether.setTarget(A)`/`setRepel(A)`/ `setAttract(A)` method on the appropriate players `Ship.` Because every object is in visible scope to the `Environment` class through the `Array` containers, there is sufficient information available. We anticipate this could be a slow operation (especially when there are many objects on the screen and in the `Array` containers).
The `Environment.onEnterFrame` method is responsible for (i) calling the `updateVelocity()` methods of all `Ship` objects. (ii) performing collision detection and resolution on *all objects in scope* (iii) calling the of `updatePosition()`methods of all `Ships, Spices,` and `Asteroids.`

Task(ii) is conceptually straightforward but involves some programming effort. Resolving any given collision can involve: (a) exchange of momentum, (b) object deletion, (c) adjusting player score, and so forth. We envision the following policies:

- `Ship` to `Ship` collision: exchange of momentum, destruction is possible if `Ships` are traveling sufficiently fast. A destroyed ship returns near its HomeBase with a score penalty.
- `Ship` to `Asteroid`: impart momentum to `Asteroid`, destroy `Ship`.
- `Asteroid` to `Asteroid, Spice` to `Spice`: exchange of momentum
- `Asteroid` to `Spice`: exchange of momentum
- `Ship` to `Spice`: don't even check. `Ships` can only interact with spice by tethering it and dragging it to `HomeBase`
- `Spice` to `HomeBase`: destroy `Spice`, update player score. Maybe make some new `Spice` elsewhere in the world to avoid exhausting it all.

Collision detection could also be a potentially slow operation if a large number of objects are present in the universe.

Wrapping Up
Clearly this document does not address all the ultimate features of the game (there is no mention of time limit, rounds, upgrading, sound effects, music). We consider this class specification to be the bare minimum playable requirements, once it is satisfied we anticipate there is little chance of project failure. If it is implemented correctly, additional features can be added incrementally with low additional risk.

Utilities
Here is a list of coding utilities we either will or are considering using followed by some details, notes, etc.

Text editors – Notepad, SE|PY

Any text editor works just fine as long as the resultant file has the .as extension.

Compiler – MTASC

As mentioned above, we'll be using MTASC since the alternative, MMC, is slow.

Debugging – Flash Debug Player

This player should be provided with Flash and is installed in place of the Flash Player. Most simply, swf files run under the Debug Player and debugging information is output to a log file. It can also be used in conjunction with the Flash IDE for better results (probably).

Testing/Profiling – our own unit tests, (Flex?)

Apparently, Macromedia Flex has powerful coding utilities for profiling, unit testing and such. It's not installed on the ACCAD machines, but there is a trial version. Flash Professional 8 provides only a simple bandwidth profiler which could be useful. Otherwise, we will have to write our own tests, probably in accordance with the XP style for projects as it is more suitable for smaller groups.

Graphics/Sound Processing – swfmill + Chad's program

To compile the image SWF libraries, we can use swfmill in addition to Chad's program (which bypasses the need to write XML files) to process the graphics. As far as sound goes, we'll grab them dynamically, directly from a directory.