Technical Manual – Contents

1. Purpose of this Document

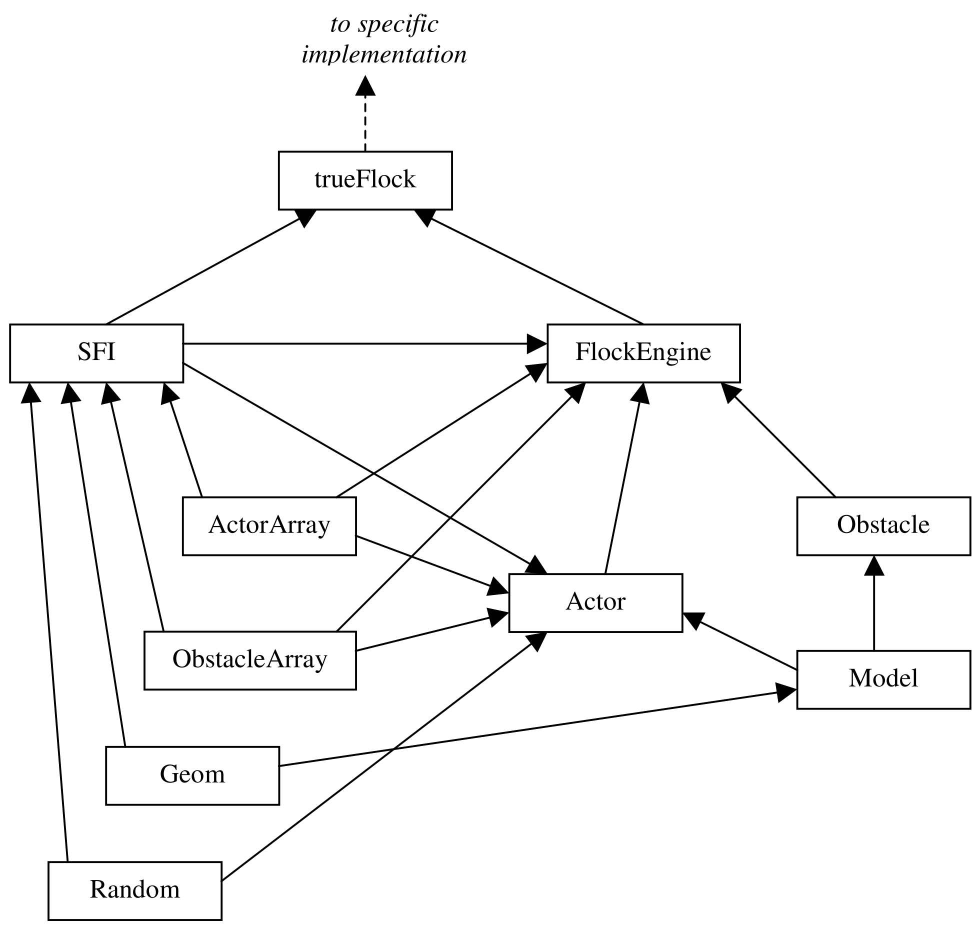
This document serves as an extension to the Design Specifications document. It describes the project's functionality in general and also gives a detailed description of all of the classes and methods which compose the project. A detailed explanation of the specific Softimage implementation of the project is also included.

2. Overall Design

The Design Specifications stated that although an object-oriented language (C++) and object-oriented concepts (classes) were being used, a formal object-oriented design was not necessary. It was argued that the real "goodies" of the object-oriented paradigm were not going to be used, such as polymorphism and inheritance.

As the project progressed, it became more readily apparent that, indeed, the prizes of object-oriented design would prove useful and, in some cases, essential. From that point on, a more rigid adherence to object-oriented principles was observed.

The following diagram shows the classes which make up the project. The links between classes indicate their relationships and dependencies. The arrow from one class to another can be read "provides functionality to."



Before proceeding to detail the classes and actual methods of the project, a summary of the tool's workings is provided: The main control of the project comes from the FlockEngine class. The constructor method of this class performs initialization such as creating arrays of the Actor and Obstacle objects in the scene. The Run() method then iterates for each frame of the animation. During each frame, each Actor is allowed to choose its course based on its "instincts". These are the programmed behaviours which control the Actor's movements. To summarize, the six instincts are:

1. Flock – fly in a flock formation
2. Avoid Obstacles – avoid colliding into Obstacle models
3. Remain In Universe – remain inside the universe (a sphere of user-specified radius)
4. Follow Script – fly along a user-selected path
5. Remain On Course – remain on previous course
6. Fly At Will – fly in a random direction

The result of each of these instincts is a vector which points in the direction that particular instinct deems best for the Actor. For example, the Flock instinct will produce a vector towards the "center" of the flock while the Avoid Obstacles instinct will return a vector pointing away from a nearby obstacle. The resultant vectors are scaled according to priorities set by the user and combined to attain an InstinctResult vector for that particular Actor, for the current frame. And so the loop continues, each Actor decided where to fly for each frame, based upon its local perceptions of the universe.

The project has been designed so as to be easily portable to another animation system and/or operating system. In fact, this project has produced two implementations. The primary one is for Softimage|3D on Silicon Graphics workstations. The second runs under Windows 95 or Windows NT using OpenGL for its graphical output.

When porting the project to another system, only a single class needs to be modified: the SFI (Softimage-Flocking Interface) class. This class acts as an interface between the flocking code and the external environment. That is, one only needs to "fill in" the members of the SFI class with code specific to the system being ported to. None of the other classes should need to be changed.

To implement the project, a trueFlock object is instantiated and the tool executes. It is up to the specific implementation on how the output from the tool is handled. The information for each frame can be displayed in real-time or buffered until all frames have been computed. Again, it is thanks to the design of the SFI class that this flexibility can be provided.

3. Description of classes

The classes are given in reverse-order of dependency: Classes which solely provide functionality for other classes are listed first. Classes at the top of the hierarchy which provide functionality to the external environment are listed last.

Random

This class generates pseudo-random numbers as required by other classes of the project.

Derived from: None
Provides functionality to: Actor, SFI
Receives functionality from: None

Attributes:

None

Methods:

CRandom (constructor, public)

```
CRandom::CRandom (void)
```

When an object of this class is instantiated, a random seed is generated using routines from the C standard library based upon the current system time.

Generate (public)

```
double CRandom::Generate (double Min, double Max)
```

Min – minimum value that can be generated and returned
Max – maximum value that can be generated and returned

After instantiating a CRandom object, this method is called to return a pseudo-random number between the given minimum and maximum boundaries.

point3D

Class which describes a point in 3-space and provides operations for points.

Derived from: None
Provides functionality to: vect3D, Model, Obstacle, Actor, FlockEngine, SFI
Receives functionality from: None

Attributes:

x, **y**, **z** (double, public)

Holds the x, y and z coordinates of this point. These attributes are publicly accessible for setting and retrieving so no accessor methods are required. This is because use of these attributes is so common.

Methods:

point3D (constructor, public)

```
point3D::point3D (void)
```

Creates a point3D object initialized to the origin (0, 0, 0).

point3D (constructor, public)

```
point3D::point3D (double xx, double yy, double zz)
```

xx, *yy*, *zz* – coordinate values

Creates a point3D object initialized with the given coordinates (xx, yy, zz).

ScalarMultiply (public)

```
void point3D:: ScalarMultiply (double scalar)
```

scalar – scalar multiple

Multiples this point by the given scalar.

Add (public)

```
void point3D::Add (point3D What)
```

What – another point

Adds the given point to this one.

DistanceTo (public)

```
double point3D::DistanceTo (point3D Other)
```

Other – another point

Returns the distance between the given point and this one. This distance is calculated by:

$$\mathrm{sqrt}(\ (\mathrm{Other.x} - \mathrm{this.x})^2 + (\mathrm{Other.y} - \mathrm{this.y})^2 + (\mathrm{Other.z} - \mathrm{this.z})^2\ )$$

vect3D

Class which describes a vector in 3-space. A vector has direction and magnitude, usually expressed as two points: an initial point and a terminal point. Here, all vectors have initial point at the origin (0, 0, 0). Therefore, only the terminal point need be considered.

Derived from: None
Provides functionality to: Model, Actor, FlockEngine, SFI
Receives functionality from: point3D

Attributes:

x, **y**, **z** (double, public)

The coordinates of this vector's terminal point. Publicly accessible for easy and common access.

Methods:

vect3D (constructor, public)

```
vect3D::vect3D (void)
```

Creates a vector with terminal point initialized to (0, 0, 0).

vect3D (constructor, public)

`vect3D::vect3D (double xx, double yy, double zz)`

xx, *yy*, *zz* – coordinate values

Creates a vector with terminal point initialized to the given coordinates (xx, yy, zz).

Create (public)

`void vect3D::Create (point3D From, point3D To)`

From – initial point
To – terminal point

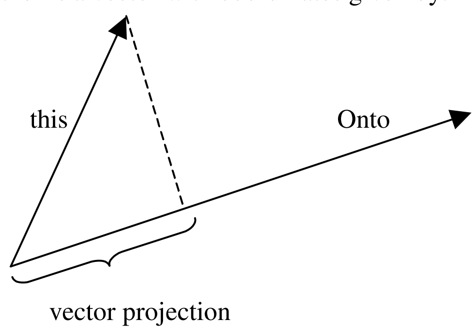
Create a vector from the given points and standardize it such that its initial point lies at the origin (0, 0, 0).

VectorProjection (public)

`vect3D vect3D::VectorProjection (vect3D Onto)`

Onto – vector

Calculate and return the vector projection of this vector onto the given one. The vector projection of one vector onto another is a vector with coordinates given by:



this × Onto = (this • Onto) / (this • this) * this

DotProduct (public)

`double vect3D::DotProduct (vect3D With)`

With – vector

Calculate and return the dot product of this vector with the given one. The dot product of two vectors is a number calculated by:

this • With = this.x * With.x + this.y * With.y + this.z * With.z

ScalarMultiply (public)

`void vect3D::ScalarMultiply (double scalar)`

scalar – scalar multiple

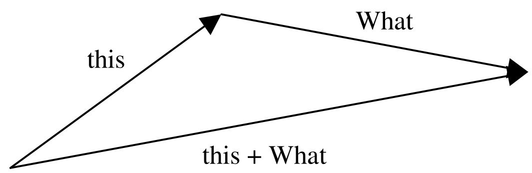
Multiply this vector by the given scalar.

Add (public)

```
void vect3D::Add (vect3D What)
```

What – vector

Add the given vector to this one.



Add (public)

```
void vect3D::Add (point3D What)
```

What – point

Same as adding vectors. Simplifies having to create a vector from the given point and using the previous method.

Clear (public)

```
void vect3D::Clear (void)
```

Reset this vector to the origin (0, 0, 0).

ToPoint (public)

```
point3D vect3D::ToPoint (void)
```

Convert this vector into a point by returning its terminal point.

Magnitude (public)

```
double vect3D::Magnitude (void)
```

Return the magnitude (length) of this vector. The magnitude is calculated by:

$|this| = sqrt(this.x^2 + this.y^2 + this.z^2)$

AngleWith (public)

```
double vect3D::AngleWith (vect3D With)
```

With – vector

Return the angle between this vector and the given one. The angle, θ, between two vectors can be calculated using the equation:

$\cos(\theta) = (this \bullet With) / (|this| * |With|)$

Print (debugging, public)

```
void vect3D::Print (char *Message)
```

Message – pointer to null-terminated character string

Prints the given message as a prompt and then writes this vector's terminal point to the standard output file. This method is only used for debugging and testing purposes.

Model

Traditionally, animation packages use the term “model” to describe the objects which are manipulated in order to create the animation. This class represents such an object which can be animated in the scene. This object can range from a simple, single point to a complex hierarchy of primitive shapes.

This class serves as the superclass for the Actor and Obstacle classes. This base class provides the essential attributes and methods which are common to both of these types of animation objects.

Originally, the Design Specifications indicated that object-oriented “goodies” would not be necessary for this project. This class is an example of the fallacy of that original presumption. The similarities between Actors and Obstacles begs for inheritance- one of the advantages of object-orientation.

Another advantage of this approach is to allow passing of an Actor or Obstacle object to be handled using polymorphism. That is, the receiving function will take a Model object and deal with the attributes common to both Actors and Obstacles without particular knowledge of which type of object is actually being passed. Of course, polymorphism is another strong point of the object-oriented paradigm.

Derived from: None
Gives functionality to: Actor, Obstacle
Receives functionality from: point3D

Attributes:

CurrentPos (point3D, protected)

Holds the position of this model in the animation scene for the current frame. Since this member is protected, accessor methods are provided (see below).

PrevPos (point3D, protected)

Holds the position of this model in the animation scene for the previous frame. Since this member is protected, accessor methods are provided (see below).

ModelRef (void *, protected)

This attribute allows a specific piece of data to be attached to each model. This attribute can hold a single integer as an index into an array or can be a pointer to a structure containing more detailed information about this particular model.

Methods:

GetCurrentPos (public)

```
point3D Model::GetCurrentPos (void)
```

Returns the position of this model in the animation scene for the current frame. This is an accessor method for the protected CurrentPos attribute.

SetCurrentPos (public)

```
void Model::SetCurrentPos (point3D NewCurrent)
```

NewCurrent – a point

Sets the position of this model in the animation scene for the current frame. This is an accessor method for the protected CurrentPos attribute.

GetPrevPos (public)
```
point3D Model::GetPrevPos (void)
```

Returns the position of this model in the animation scene for the previous frame. This is an accessor method for the protected PrevPos attribute.

SetPrevPos (public)
```
void Model::SetPrevPos (point3D NewPrev)
```
NewPrev – a point

Sets the position of this model in the animation scene for the previous frame. This is an accessor method for the protected PrevPos attribute.

GetModelRef (public)
```
void * Model::GetModelRef (void)
```

Returns the specific reference data for this model. This is an accessor method for the protected ModelRef attribute.

SetModelRef (public)
```
void Model::SetModelRef (void * NewModel)
```
NewModel – pointer to specific data about this model

Sets the specific reference data for this model. This is an accessor method for the protected ModelRef attribute.

ObstacleArray

This class represents an array of Obstacle objects. Methods are provided for adding and retrieving elements, etc. Note that this class does not need to know exactly what makes up an Obstacle object since it only deals with references (pointers) to these objects.

Derived from: None
Provides functionality to: Actor, FlockEngine, SFI
Receives functionality from: None

Attributes:

m_pData (Obstacle **, protected)
Pointer to the beginning of the array. Each array element consists only of a pointer to an Obstacle object. Since this member is protected, accessor methods are provided (see below).

m_nSize (int, protected)
Holds the number of elements currently in the array. Since this member is protected, accessor methods are provided (see below).

Methods:

ObstacleArray (constructor, public)
```
ObstacleArray::ObstacleArray (void)
```

Creates an array of Obstacle objects. Initially, the array is empty and the number of elements is set to zero.

ObstacleArray (constructor, public)

```
ObstacleArray::ObstacleArray (int Size)
```

Size – integer for initial size of array

Creates an array of Obstacle objects. Initially, enough space is allocated to hold the given number of elements. However, all elements remain uninitialized.

ObstacleArray (deconstructor, public)

```
~ ObstacleArray::ObstacleArray (void)
```

Destroys this array object. If any elements have been allocated, they are freed. The number of elements is reset to zero.

SetSize (public)

```
void ObstacleArray::SetSize (int Size)
```

Size – integer size of array

Allocates space to hold the given number of elements. This method should only be called if the array was previously empty. If memory allocation is successful, the protected m_nSize attribute is set to Size.

GetSize (public)

```
int ObstacleArray::GetSize (void)
```

Returns the current size of this array. This is an accessor method for the protected m_nSize attribute. If the array is empty, zero is returned.

Get (public)

```
Obstacle * ObstacleArray::Get (int Index)
```

Index – array index

Retrieves the array element at the given index. If the array is empty or the index is out of bounds, NULL is returned. Otherwise, a pointer to that Obstacle is returned.

Set (public)

```
void ObstacleArray::Set (int Index, Obstacle * Elt)
```

Index – array index
Elt – pointer to Obstacle object

Puts the given Obstacle into the array at the given index. If the array is empty, or the index is out of bounds, the array remains unmodified. Otherwise, the array element is set equal to the given pointer.

ActorArray

This class represents an array of Actor objects. Methods are provided for adding and retrieving elements, etc. Note that this class does not need to know exactly what makes up an Actor object since it only deals with references (pointers) to these objects.

Derived from: None
Provides functionality to: Actor, FlockEngine, SFI
Receives functionality from: None

Attributes:

m_pData (Actor **, protected)

Pointer to the beginning of the array. Each array element consists only of a pointer to an Actor object. Since this member is protected, accessor methods are provided (see below).

m_nSize (int, protected)

Holds the number of elements currently in the array. Since this member is protected, accessor methods are provided (see below).

Methods:

ActorArray (constructor, public)

```
ActorArray::ActorArray (void)
```

Creates an array of Actor objects. Initially, the array is empty and the number of elements is set to zero.

ActorArray (constructor, public)

```
ActorArray::ActorArray (int Size)
```

Size – integer for initial size of array

Creates an array of Actor objects. Initially, enough space is allocated to hold the given number of elements. However, all elements remain uninitialized.

ActorArray (deconstructor, public)

```
~ ActorArray::ActorArray (void)
```

Destroys this array object. If any elements have been allocated, they are freed. The number of elements is reset to zero.

SetSize (public)

```
void ActorArray::SetSize (int Size)
```

Size – integer size of array

Allocates space to hold the given number of elements. This method should only be called if the array was previously empty. If memory allocation is successful, the protected m_nSize attribute is set to Size.

GetSize (public)

```
int ActorArray::GetSize (void)
```

Returns the current size of this array. This is an accessor method for the protected m_nSize attribute. If the array is empty, zero is returned.

Get (public)

```
Actor * ActorArray::Get (int Index)
```

Index – array index

Retrieves the array element at the given index. If the array is empty or the index is out of bounds, NULL is returned. Otherwise, a pointer to that Actor is returned.

Set (public)

```
void ActorArray::Set (int Index, Actor * Elt)
```

Index – array index
Elt – pointer to Actor object

Puts the given Actor into the array at the given index. If the array is empty, or the index is out of bounds, the array remains unmodified. Otherwise, the array element is set equal to the given pointer.

Obstacle

For this project, there are two types of models in the animation scene: Actors and Obstacles. Actor models are those that the user chooses to be members of the flock. Obstacle models are models the user wishes to be considered obstructions in the universe. The Actors should avoid colliding with all Obstacles. This is taken care of with the Actor's "Avoid Obstacles" instinct (see below).

This class represents Obstacle models in the scene. The position of Obstacles is not modified by the flocking tool. However, the Obstacles do not have to remain stationary. During each frame, the position of each Obstacle model can be retrieved from the animation scene so that the Actors will know where to avoid flying.

It is up to methods of the SFI class to initialize an array of Obstacle objects and change their positions for each frame, if desired.

Note that because Obstacles are simply objects to be avoided and don't have any will of their own, this class is considerably less complicated than the Actor class.

Derived from: Model
Provides functionality to: FlockEngine
Receives functionality from: point3D, Model

Attributes:

None

Methods:

Obstacle (constructor, public)

```
Obstacle::Obstacle (point3D Position, void * Reference)
```

Position – initial position of this Obstacle
Reference – pointer to data specific to this Obstacle

Creates an Obstacle object. The object's CurrentPos and PrevPos attributes (derived from the Model class) are both set to the given position. The ModelRef attribute is set to the given pointer. This reference can hold more information about this particular Obstacle as the programmer desires.

Obstacle (deconstructor, public)

```
~ Obstacle::Obstacle (void)
```

Destroys this Obstacle object.

Actor

For this project, there are two types of models in the animation scene: Actors and Obstacles. Actor models are those that the user chooses to be members of the flock. Obstacle models are models the user wishes to be considered obstructions in the universe. The Actors should avoid colliding with all Obstacles. This is taken care of with the Actor's "Avoid Obstacles" instinct (see below).

This class represents Actor models in the scene. Actors are the most important objects in this project. Their "instincts" are what create the animation. During each frame, the Flocking Engine asks each Actor where it would like to fly. Each Actor makes its decision based upon the results of six fundamental instincts. They are:

1. Flock – fly in a flock formation
2. Avoid Obstacles – avoid colliding into Obstacle models
3. Remain In Universe – remain inside the universe (a sphere of user-specified radius)
4. Follow Script – fly along a user-selected path
5. Remain On Course – remain on previous course
6. Fly At Will – fly in a random direction

Each instinct observes its immediate surroundings and calculates a course that would best satisfy that instinct. This resultant course is a vector. These vectors are scaled and combined by the Flocking Engine to arrive at the final decision vector for the current Actor, for the current frame.

Derived from: Model
Provides functionality to: FlockEngine
Receives functionality from: Random, point3D, vect3D, Model, ObstacleArray, ActorArray, SFI

Attributes:

InstinctResult (vect3D, public)
When intermediate vectors are calculated for each instinct, the running sum is placed here.

FlockElt (C-style structure)
C-style structure containing two fields:
double Distance – distance from current Actor to this one
Actor * pActor – pointer to Actor object
This structure is used in creating an array of nearby flockmates for the I_Flock method. An old, C-style declaration was necessary because the method uses the C standard library qsort() function to sort the array.

Methods:

Actor (constructor, public)

```
Actor::Actor (point3D Position, void * Reference, SFI * SFIPtr)
```

Position – initial position of this Actor
Reference – pointer to data specific to this Actor
SFIPtr – pointer to SFI object

Creates an Actor object. The object's CurrentPos attribute (derived from the Model class) is set to the given position. The PrevPos attribute (also derived from the Model class) is generated randomly. It is generated such that its distance from CurrentPos is equal to the maximum acceleration value set by the user. The pointer to the SFI object allows this method to retrieve this value. This gives the Actor a random initial heading. The ModelRef attribute is set to the given pointer. This reference can hold more information about this particular Actor as the programmer desires.

Actor (deconstructor, public)

```
~ Actor::Actor (void)
```

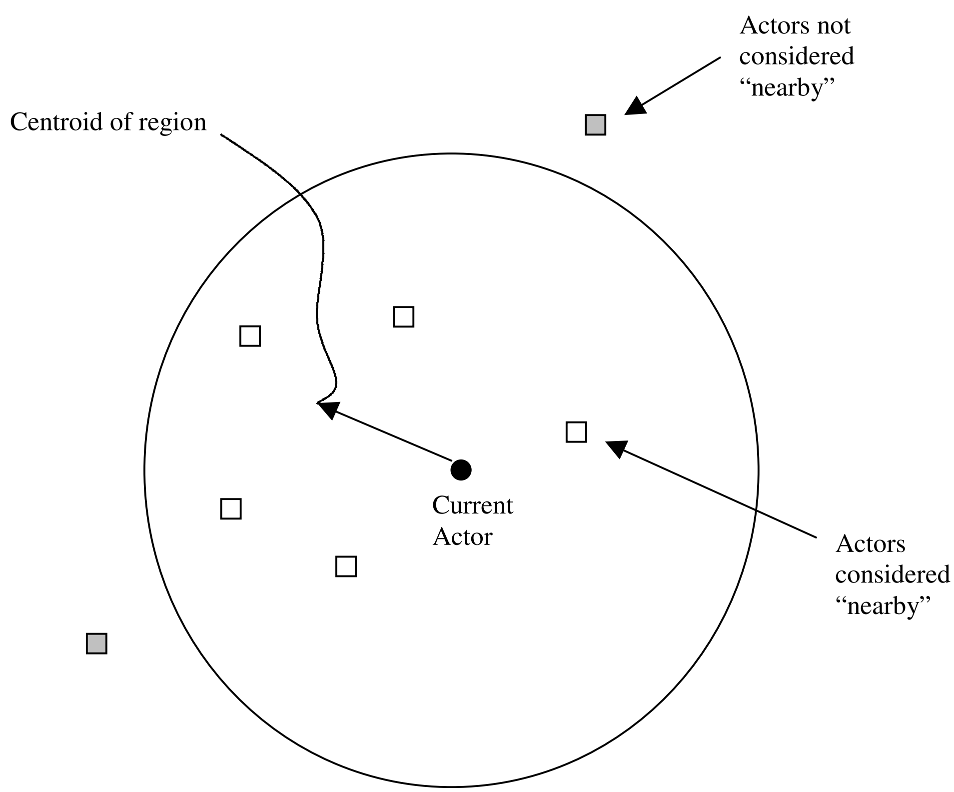
Destroys this Actor object.

I_Flock (public)

```
vect3D Actor::I_Flock (int NumActor, SFI * SFIPtr, ActorArray *
                      pActorArray)
```

NumActor – integer count of Actors
SFIPtr – pointer to SFI object
pActorArray – pointer to array of Actors

This is the primary instinct. It controls the Actor's desire to flock with the other Actors.



To calculate the result of this instinct, first a region of "nearby" flockmates is created. The definition of which Actors lie in this region depends upon the user. There are three choices of defining this region:

1. All Actors within a sphere of specified radius are considered "nearby."
2. A certain number of Actors (just how many is specified by the user) which are closest to this one are considered "nearby."
3. A combination of the above two options. That is, the Actors within a sphere of specified radius up to a given maximum are considered "nearby."

Next, the centroid of this region is calculated by summing the current positions of all of the Actors which lie within it. Normally, a vector is plotted from this Actor's current position to the centroid and returned as this instinct's result.

However, a check is performed to see if the flock is too "bunched up." That is, the distance between this Actor and its nearest neighbour is examined. If this distance is less than a specified threshold, the result of the instinct is a vector which points away from the closest Actor. This threshold value is controlled by the user-specified value of the Flock Density option.

This method consults the SFI object to retrieve options set by the user for this instinct.

I_AvoidObstacle (public)

```
vect3D Actor::I_AvoidObstacle (int NumObstacle, SFI * SFIPtr,
                    ObstacleArray *pObstacleArray)
```

NumObstacle – integer count of Obstacles
SFIPtr – pointer to SFI object
pObstacleArray – pointer to array of Obstacles

This instinct controls the Actor's desire to not collide with Obstacles in the scene. The Obstacles are models selected by the user. These models are not restricted to one location, they can be animated as well and the Actors will try to avoid them.

Collision detection and avoidance are major topics in Computer Science. Entire projects could be devoted to them exclusively. In order to keep things simple, a rather simple algorithm is implemented here: the "force field" model.

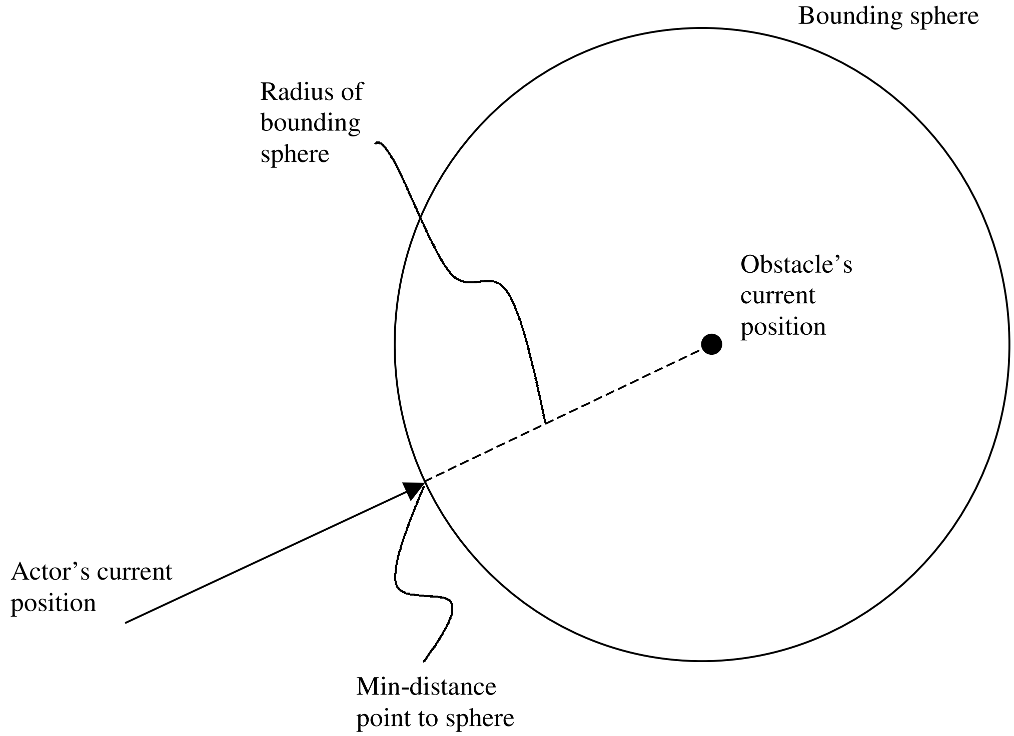
Each Obstacle is surrounded by a "force field" which is the minimum-volume sphere which encompasses the entire model. An Actor who is trying to avoid the Obstacle plots a vector from its current position to the nearest point on the Obstacle's "bounding sphere." This vector is inverted to point away from the Obstacle and is scaled in proportion to its magnitude.

The result is a simplified version of how forces like gravity and magnetism actually work in the real world. One problem is that some models may not lend themselves well to a spherical field. That is, if the user created a long rectangle as an Obstacle, the bounding sphere of the model would not accurately simulate its volume. It should be easy to insert a more sophisticated collision avoidance algorithm in the future, to rectify this shortcoming.

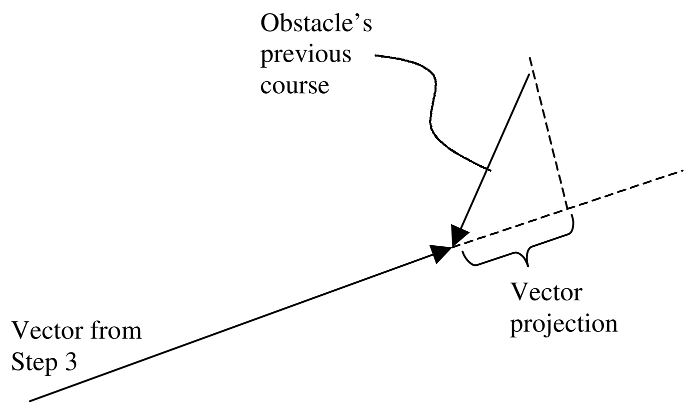
The algorithm for this method is as follows:

1. Loop for each Obstacle within this Actor's "vision." The range of vision is a distance set by the user so that only Obstacles within a sphere of this radius will be considered in the calculation of a collision-avoiding vector.

2. Obtain the bounding sphere of the Obstacle. Calculate a vector from the Actor's current position to the minimum-distance point on the bounding sphere. This can be calculated by finding the distance between the Actor's and Obstacle's current positions and subtracting the radius of the bounding sphere.



3. This vector can be modified slightly by taking the Obstacle's movement into account: Plot a vector from the Obstacle's previous position to its current one, and find the vector projection of this vector onto the one found in Step 3. By adding this projection to the original vector, the instinct is made a little "smarter." That is, if the Obstacle is moving away from the Actor, the resultant vector will become larger. If the Obstacle is moving towards the Actor, the result will become smaller.



4. The vector is inverted and scaled proportionally to the square of its magnitude. This means if the distance to the Obstacle is small, its importance will be exaggerated. If the Obstacle is far away, the resultant vector will be small.

I_RemainUniverse (public)

```
vect3D Actor::I_RemainUniverse (SFI * SFIPtr)
```

SFIPtr – pointer to SFI object

This instinct causes the Actor to remain inside of the universe. The universe, in this case, is a sphere of user-specified radius. The purpose of this instinct is to keep the Actors within the camera's view.

The implementation of this instinct is somewhat similar to the Avoid Obstacles instinct. A vector is plotted from the Actor's current position to the minimum-distance point on the universal sphere. This vector is inverted and scaled according to its magnitude.

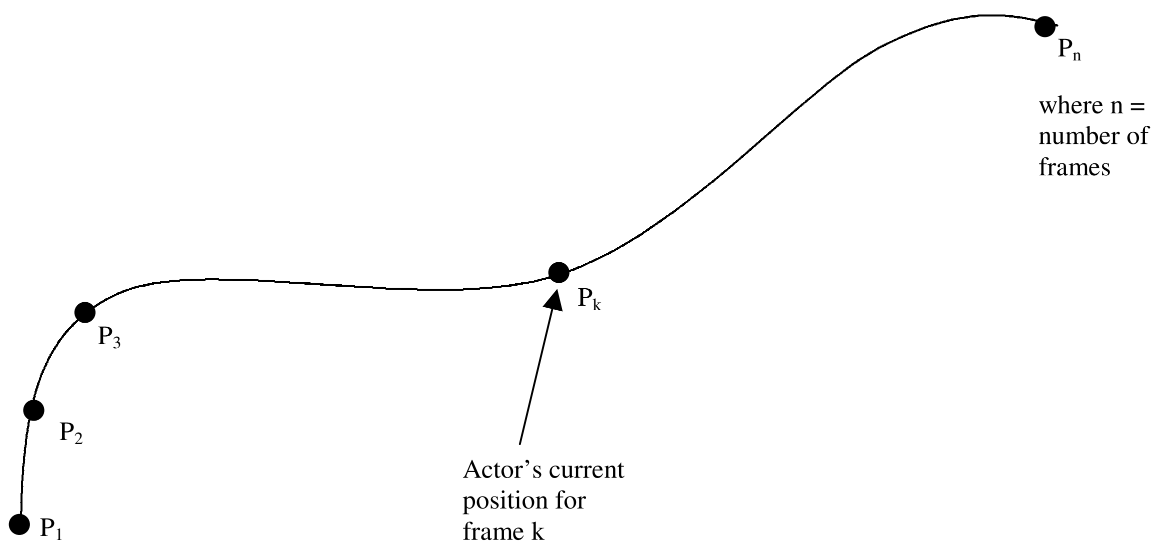
I_FollowScript (public)

```
vect3D Actor::I_FollowScript (point3D ScriptPoint, SFI * SFIPtr)
```

ScriptPoint – point on script path for current frame
SFIPtr – pointer to SFI object

This instinct gives the user a little more control over the animation created. The user can pick a path (spline, curve, etc.) that the flock will attempt to follow over the course of the animation.

Technically, this instinct is contrary to "real flocking" in that it destroys some of its autonomous, decentralized nature. However, since the end result is an animation and, in many cases, the user would like an animation which looks the way he or she envisions it, this instinct is actually a very useful tool.



The implementation of this method is simple: The selected path is segmented into a number of distinct points equal to the number of frames which will compose the animation. Mathematically, the path P = $\{p_i \mid$ for i = 1 to the number of frames in the animation$\}$. During each frame, each Actor plots a vector from its current position to the point on the script path for the current frame.

I_RemainCourse (public)

```
vect3D Actor::I_RemainCourse (void)
```

This instinct simply persuades the Actor to continue along its previous course. It is included to give, at the very least, a crude approximation of momentum. Without it, the Actor's movement may seem (more) unrealistic.

The result of this instinct is a vector created from this Actor's previous position to its current one.

I_FlyWill (public)

```
vect3D Actor::I_FlyWill (void)
```

This instinct adds an element of randomness and uncertainty to the flock's motion.

A random coordinate on the unit sphere is calculated and a vector created with it as its terminal point.

ActorCanSee (private)

```
int Actor::ActorCanSee (point3D ObstPos, SFI * SFIPtr)
```

ObstPos – Obstacle's current position
SFIPtr – pointer to SFI object

This method returns whether the given Obstacle is within the Actor's range of sight. That is, the SFI is consulted for the vision range and is compared to the distance between the Actor's current position and the given point. A non-zero return value means the Actor can see the Obstacle while returning zero indicates the Actor cannot see the Obstacle.

GetCollisionCourse (private)

```
vect3D Actor::GetCollisionCourse (point3D ObstPos, double Radius)
```

ObstPos – Obstacle's current position
Radius – radius of bounding sphere

This method is called by the I_AvoidObstacle() method. It returns a vector from the Actor's current position to the minimum-distance point on the Obstacle's bounding sphere. This code is similar to the code to find the minimum-distance point to the universal sphere for the I_RemainUniverse() method.

CompareFlockElts (C-style function)

```
int CompareFlockElts (const void *a, const void *b)
```

a – pointer to FlockElt structure
b – pointer to FlockElt structure

This is a comparison function passed to the C standard library function qsort(). It is used by the I_Flock method to sort the array of nearby flockmates. Its return values can be summarized as follows:

> 0 – a's distance is greater than b's distance
< 0 – a's distance is less than b's distance
= 0 – a's distance is equal to b's distance

FlockEngine

The Flocking Engine is, as it sounds, in charge of the entire project. It initializes everything by creating the appropriate number of Actor and Obstacle objects. It creates the animation by asking each Actor for the results from its instincts for each frame. It also cleans up and frees any memory that was allocated.

Derived from: None
Provides functionality to: trueFlock
Receives functionality from: point3D, vect3D, ObstacleArray, ActorArray, Obstacle, Actor, SFI

Attributes:

m_SFIPtr (SFI *, private)
Pointer to SFI object needed to access user's selected options, etc. This attribute is private because it is not accessed outside of this class.

NumActor (int, private)
Count of the number of Actors in the scene. This attribute is private because it is not accessed outside of this class.

NumObstacle (int, private)
Count of the number of Obstacles in the scene. This attribute is private because it is not accessed outside of this class.

m_pActorArray (ActorArray *, private)
Pointer to the array of Actor objects. This attribute is private because it is not accessed outside of this class.

m_pObstacleArray (ObstacleArray *, private)
Pointer to the array of Obstacle objects. This attribute is private because it is not accessed outside of this class.

CurrentFrame (int, private)
Loop counter for the current frame being computed. This attribute is private because it is not accessed outside of this class.

NumFrame (int, private)
The number of frames which will make up the animation. This attribute is private because it is not accessed outside of this class.

ScriptPointList (point3D *, private)
Pointer to an array of points. These are the sampled points from the path selected by the user for the Follow Script instinct. Of course, if not path is selected, this array will be empty. This attribute is private because it is not accessed outside of this class.

Methods:

FlockEngine (constructor, public)

```
FlockEngine::FlockEngine (SFI * SFIPtr)
```

SFIPtr – pointer to SFI object

Creates a FlockEngine object and initializes it. All private attributes are initialized with values set by the user. The SFIPtr parameter is copied to the m_SFIPtr attribute for access by other members of the class. The arrays of Actors and Obstacles are also created and initialized. If the user selected a path for the Actors to follow, it is segmented and its points are put into the ScriptPointList array.

FlockEngine (deconstructor, public)

```
~ FlockEngine::FlockEngine (void)
```

Destroys this FlockEngine object. All of the Actor and Obstacle objects are destroyed and their arrays are freed. If a script path was selected by the user, the ScriptPointList array is also deallocated.

Run (public)

```
void FlockEngine::Run (void)
```

This is the main method of the entire project. It is the starting point for all primary execution. It consists of a loop which iterates for each frame of the animation. During each frame, the instinctive result for each Actor is computed and stored. The positions of each Actor and Obstacle are also updated during each frame.

Inside the loop, a call to FindInstinctResults() is made to retrieve the resultant vectors for each of the current Actor's six instincts. These vectors are scaled according to their priorities, rotated according to a maximum turning angle and truncated to a maximum length, all set by the user.

A call to the UpdateActorPos() method of the SFI class outputs the results for each Actor during each frame. It is up to the system-specific SFI, then, to decide how to handle the results. For example, they can be stored and later processed when all frames are complete or they can be drawn in real-time.

GetCurrentScriptPoint (public)

```
point3D FlockEngine::GetCurrentScriptPoint (void)
```

If the user selected a path for the Actors to follow, this method retrieves the point they should be aiming towards for the current frame. Otherwise, the result of this method will be ignored.

FindInstinctResults (private)

```
void FlockEngine::FindInstinctResults (int i)
```

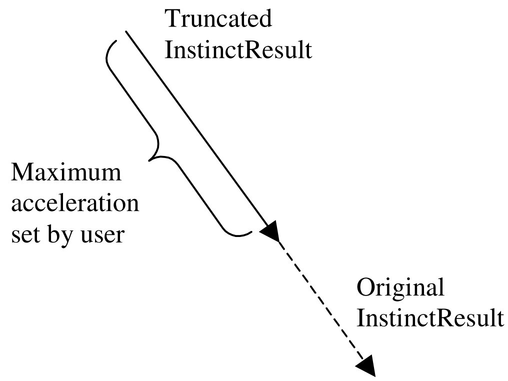
i – index of current Actor

This method retrieves results for each of the six instincts which guide an Actor's movements. The passed parameter is an index into the array of Actors (m_pActorArray). Initially, the Actor's InstinctResult attribute is reset to (0, 0, 0).

After each result is retrieved, it is scaled by the appropriate priority set by the user and added to the Actor's InstinctResult attribute. After all instincts have been taken care of, the InstinctResult attribute contains the sum of the decision vectors. Two tests are then performed.

First, the angle between the Actor's previous course and the InstinctResult vector are compared. If this angle is greater than the maximum turning angle specified by the user, the InstinctResult is rotated so that it lies at an angle equal to the maximum but in the same plane as the original vectors. The FixVector() method performs this modification. Note that the rotated vector will have the same magnitude as the original InstinctResult.

Secondly, the magnitude of the InstinctResult vector is compared to the maximum acceleration setting entered by the user. If the magnitude is greater than the maximum, the vector is truncated so that it's magnitude equals the maximum.



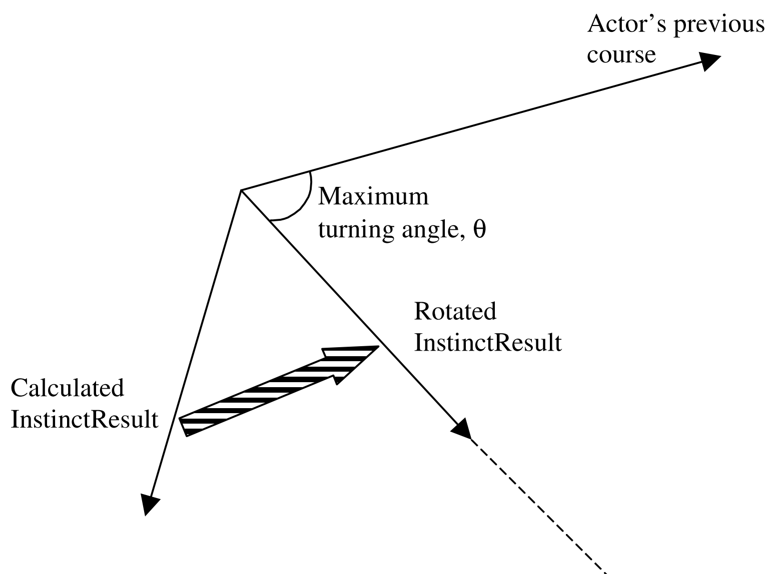
FixVector (private)

```
void FlockEngine::FixVector (Actor * pActor, vect3D C)
```

pActor – pointer to Actor object
C – vector representing Actor's previous course

If the angle between the Actor's previous course and its InstinctResult vector is greater than the user-specified maximum turning angle, the InstinctResult vector must be rotated so that it does not violate the user's wishes. This method rotates the InstinctResult vector such that:

1. The angle between the new InstinctResult and the Actor's previous course will equal the maximum turning angle.
2. The new InstinctResult will lie in the same plane formed by the old InstinctResult and previous course vectors.



The solution to this problem is not an easy one. It involves some complicated and messy mathematics. The following attempts to describe how the vector is rotated while adhering to the above two conditions.

For a detailed explanation of how the vector is rotated, please see Appendix A.

SFI

The following describes the attributes and methods found in the SFI class. The descriptions of each method try to explain its purpose in general, without regard to a specific implementation. Of course, the contents of each method will change from system to system.

The SFI class acts as an ambassador between the code which creates the flocking motion and the manner in which it manifests itself. That is, it is an interface between the flocking tool and the external environment. In most cases, this environment involves the operating system being implemented on and graphics subsystem being used to draw the tool's output.

Derived from: None
Provides functionality to: Actor, FlockEngine, trueFlock
Receives functionality from: Random, point3D, vect3D, ObstacleArray, ActorArray

Attributes:

m_pUserData (void *, private)

This attribute allows a specific implementation to pass a pointer to the SFI object. This can be used however desired. In most cases, it will provide access back to a structure containing the options set by the user and possibly provide access to storage for the tool's results.

Methods:

SFI (constructor, public)

```
SFI::SFI (void * UserDataPtr)
```

UserDataPtr – pointer to implementation-specific data

Creates an SFI object. The passed parameter is copied to the private m_pUserData attribute for access by the other members of the class.

InitActorList (public)

```
void SFI::InitActorList (ActorArray * pActorArray)
```

pActorArray – pointer to array of Actor objects

Assuming that the necessary space for the given array has already been allocated, fill in each entry with an Actor object. The objects will have to be initialized depending upon the user options selected. For example, the Actors may have to generate random initial positions or their initial positions may have to be retrieved from the scene.

InitObstacleList (public)

```
void SFI::InitObstacleList (ObstacleArray * pObstacleArray)
```

pObstacleArray – pointer to array of Obstacle objects

Assuming that the necessary space for the given array has already be allocated, fill in each entry with an Obstacle object. The object will have to be initialized using data from the animation scene.

SegmentPath (public)

```
void SFI::SegmentPath (point3D * pArray, int NumPoints)
```

pArray – pointer to array of points
NumPoints – count of number of points in array

If the user selects a path for the flock to follow (for the Follow Script instinct), the selected path must be divided into a number of discrete destination points. This method performs this segmentation.

The selected path is retrieved and divided with the computed points being placed into the given array. The necessary memory should already be allocated for the array before calling this method. Just how this path is retrieved and segmented is, of course, entirely dependent on the system being implemented for.

GetBoundSphere (public)

```
double SFI::GetBoundSphere (void * ModelRef)
```

ModelRef – reference to a model in the scene

This method determines the radius of the minimum-volume sphere which completely surrounds the given model and returns it. The model reference passed is the ModelRef attribute of an Actor or Obstacle object. How exactly this attribute is used is up to a specific implementation. Most likely, an index into an array of Actor or Obstacle references would be passed.

If a distinction between Actors and Obstacles is required, a constant value can be added to the index of one or the other. For example, the index for Obstacles could be added to a constant (e.g. 1000). The passed ModelRef could be tested and, if it is greater-than or equal to this constant value, it is known that the passed model is an Obstacle.

On the other hand, a pointer to the model could be stored and passed as the model reference. It would depend on the system being developed for.

UpdateActorPos (public)

```
void SFI::UpdateActorPos (void * ModelRef, int Frame, point3D
                    NewPos)
```

ModelRef – reference to Actor's model in the scene
Frame –frame number
NewPos – new point position for Actor

This method is called once for each Actor, for each frame. After the instinct calculations have been performed and the position of the Actor is determined for the next frame, this method is called to update the external system. The Actor's new position is given so that the system can take necessary action.

For example, the new position could be stored and later used to display the animation, or the results could be put into effect immediately for a real-time animation, etc. It is up to a specific system implementation on just how to handle these updates.

UpdateObstacleList (public)

```
void SFI::UpdateObstacleList (int NextFrame, ObstacleArray *
                    pObstacleArray)
```

NextFrame – frame number
pObstacleArray – pointer to array of Obstacle objects

To allow for moving obstacles, the position of each Obstacle model must be retrieved during each frame. This method performs this update. It copies the new position of each Obstacle into the given array, for the given frame number. Once complete, the Flocking Engine can iterate for another frame.

Note that if a system does not wish to implement moving obstacles, this method will not perform any actions.

GetInstinctPriority (public)

```
double SFI::GetInstinctPriority (int Instinct)
```

Instinct – index of one of six instincts

This method returns the priority set by the user for the given instinct.

GetUniverseRadius (public)

```
double SFI::GetUniverseRadius (void)
```

This method returns the radius of the universe as set by the user.

GetActorVision (public)

```
double SFI::GetActorVision (void)
```

This method returns the range of each Actor's vision as set by the user.

GetFlockRadius (public)

```
double SFI::GetFlockRadius (void)
```

This method returns the size of the sphere within which Actors should be considered "nearby" by the Flock instinct. This value is set by the user.

GetFlockClosest (public)

```
int SFI::GetFlockClosest (void)
```

This method returns the maximum number of Actors to be considered "nearby" by the Flock instinct. This value is set by the user.

GetFrameStart (public)

```
int SFI::GetFrameStart (void)
```

This method returns the starting frame number of the animation as set by the user.

GetFrameFinish (public)

```
int SFI::GetFrameFinish (void)
```

This method returns the ending frame number of the animation as set by the user.

GetNumActor (public)

```
int SFI::GetNumActor (void)
```

This method returns the number of Actor models in the scene as set by the user.

GetNumObstacle (public)

```
int SFI::GetNumObstacle (void)
```

This method returns the number of Obstacle models in the scene as set by the user.

GetFollowScript (public)

```
int SFI::GetFollowScript (void)
```

This method returns the Boolean value TRUE (non-zero) if the user chose the select a path for the Actors to follow. If the user did not wish to select a path, this method returns FALSE (zero).

GetMaxAccel (public)

```
double SFI::GetMaxAccel (void)
```

This method returns the maximum acceleration value as set by the user. This option is the maximum distance each Actor can travel during a single frame.

GetMaxAngle (public)

```
double SFI::GetMaxAngle (void)
```

This method returns the maximum turning angle as set by the user. This is the maximum angle between an Actor's previous course and the vector result of its instincts during a single frame.

GetFlockDensity (public)

```
double SFI::GetFlockDensity (void)
```

This method returns the flock density option as set by the user. This setting controls how close the Actors may come to one another. A larger value means a sparser flock.

TestNewPosition (private)

```
int SFI::TestNewPosition (point3D Pos)
```

Pos – point position of Actor model

This method is used internally by the SFI class. It is called from the InitActorList() method. If the user chose to randomly generated the initial positions of each Actor, this method is called after each such position is generated. It returns the Boolean value TRUE (non-zero) if the new position lies outside the bounding spheres of all Obstacle models. Otherwise, it returns FALSE (zero) to indicate that a new point must be randomly generated.

trueFlock

This class forms the top of the hierarchy of classes for this project. To execute the flocking tool, the system need only instantiate an object of this class and the animation will be generated. It is the only object that the external system needs to be aware of. This completes the separation of the flocking code from the system-dependent code.

Derived from: None
Provides functionality to: External system
Receives functionality from: FlockEngine, SFI

Attributes:

None

Methods:

trueFlock (constructor, public)

```
trueFlock::trueFlock (void * UserDataPtr)
```

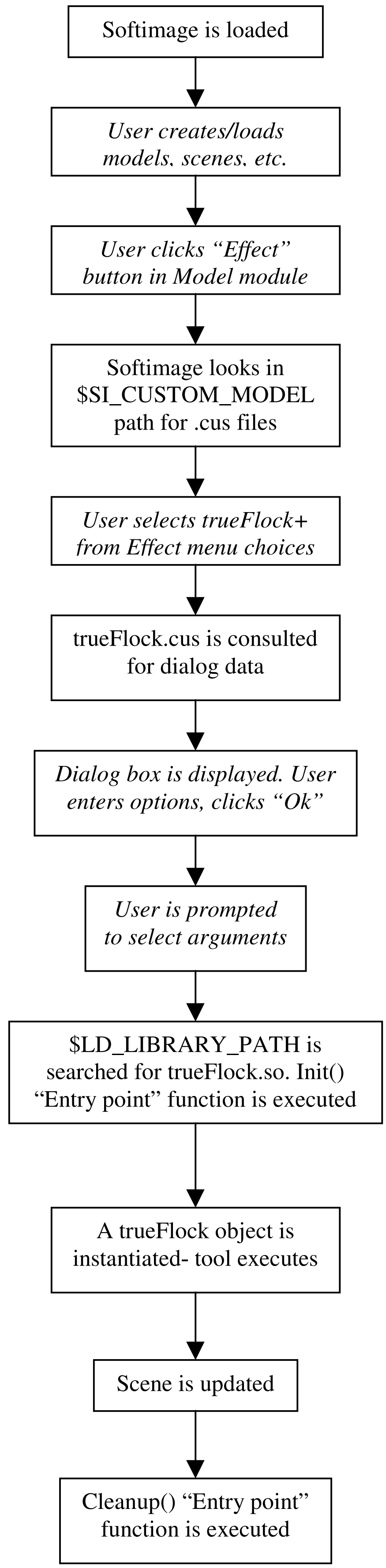
UserDataPtr – pointer to data specific to this system

Creates a trueFlock object to create a flocking animation. This method instantiates an SFI and a FlockEngine object. The UserDataPtr parameter is passed to the SFI constructor to facilitate communication between the external system and the internal flocking component. A pointer to the instantiated SFI object is passed to the FlockEngine constructor to allow the flocking code to ask for the options and data set by the user.

4. Softimage/SAAPHIRE

This section describes how the flocking portion of this project was implemented on a specific system. The primary implementation for this project was for the Softimage animation package by Microsoft Corporation. Programmers can create “custom tools” for Softimage using SAAPHIRE- Softimage’s SDK. The SAAPHIRE library is a set of functions which allow a program to interface with most of the capabilities provided by Softimage.

The following flow chart shows how a custom effect is implemented using SAAPHIRE. Italicized text indicates user-interaction at that stage.


Softimage is loaded
User creates/loads models, scenes, etc.
User clicks "Effect" button in Model module
Softimage looks in $SI_CUSTOM_MODEL path for .cus files
User selects trueFlock+ from Effect menu choices
trueFlock.cus is consulted for dialog data
Dialog box is displayed. User enters options, clicks "Ok"
User is prompted to select arguments
$LD_LIBRARY_PATH is searched for trueFlock.so. Init() "Entry point" function is executed
A trueFlock object is instantiated- tool executes
Scene is updated
Cleanup() "Entry point" function is executed

Whenever a SAAPHIRE function is called, the $LD_LIBRARY_PATH is searched for the libSAA.so file. This is the run-time library for SAAPHIRE. The appropriate code is then executed and values returned as needed, etc.

"trueFlock.cus" and "trueFlock.so"

The custom effect requires two files: "trueFlock.cus" and "trueFlock.so".

"trueFlock.cus" is simply a text file which contains information about the dialog box to be displayed to allow the user to enter options for the effect. The Softimage SDK includes a program called "Dial" which allows the programmer to create dialog boxes for custom effects using an interactive, graphical user interface.

This file also contains the names of "entry point" functions which are executed at the appropriate moment. For example, the INIT_FUNCTION is called after the user clicks "Ok" in the dialog box and selects any arguments. The CLEANUP_FUNCTION is called after the custom effect has completed execution.

The "trueFlock.cus" file also describes the arguments that must be picked by the user after dismissing the dialog box (by clicking "Ok"). A prompt can be displayed to help the user to know what to pick. With version 1.1 of SAAPHIRE, the user is forced to pick a "result" object for each "argument" picked. This is unfortunate for arguments which do not create results but which simply modify the arguments themselves. For example, the user is asked to pick a hierarchy of Actor models, a hierarchy of Obstacle models and a path for the Actors to follow. After this, he or she must pick three more objects as "results" of the three arguments. Even though this can be satisfied by simply clicking on any object three times, it can still be annoying and confusing to the user. Note that version 1.7 of SAAPHIRE fixes this problem.

The "trueFlock.cus" file must reside in a directory pointed to by the $SI_CUSTOM_MODEL environment variable in order to work.

The "trueFlock.so" file is the compiled source code of the custom effect. It must contain the entry point functions named in the "trueFlock.cus" file. This file must reside in a directory pointed to by the $LD_LIBRARY_PATH environment variable.

"IData.h" and "trueFlock.c"

The Softimage "external environment" portion of this project is kept in a file named "trueFlock.c". The data structures used are defined in a file called "IData.h".

The "IData.h" header file contains the definition of a very important structure- the InstanceData structure. This structure is created once for each instance of the flocking tool. That is, every time the user selects Effect→trueFlock+ from the Model module, an InstanceData structure is created and the tool executes. A pointer to this structure is available to each entry point function through what is called a "custom context" (see below). This allows the entry point functions to pass data between themselves.

For this project, the InstanceData structure holds all the data needed to store and create the animation: It contains fields for all of the options set by the user to pass on to the SFI whenever asked. It also contains pointers to the arguments picked by the user to be Actors and Obstacles for internal functions contained in "trueFlock.c".

Finally, the InstanceData structure contains a pointer to a two-dimensional array of Pos structures. This array holds the calculated positions for each Actor during each frame. The UpdateActorPos() method of the SFI class stores its information in this array. Once the flocking engine completes its task, this array is used to update the positions of the corresponding models in the Softimage scene, in order to create the animation.

The following is a description of the functions found in "trueFlock.c" and their purposes:

init (entry point function)

```
_CUS_EXTERN SI_Error init (SAA_CustomContext context)
```

context – custom context for this effect (see below)

This is the initialization entry point function named as the INIT_FUNCTION in "trueFlock.cus". This function instantiates a trueFlock object and executes the tool. It then calls support functions to update the Softimage scene.

The SAA_CustomContext parameter is a structure which is passed to all entry point functions. It is also the only parameter needed. It contains fields needed by Softimage such as a pointer to the current scene, etc. One crucial field it also contains is a pointer to an InstanceData structure (as mentioned earlier). The contents of this structure are defined by the programmer to hold any data that is needed between entry point functions. The communication abilities of this structure are not solely limited to entry point functions, however. By passing a pointer to the InstanceData structure as the UserDataPtr argument of a trueFlock object, the flocking code has a connection back to the external environment. This pointer is what allows the SFI to retrieve user options and to store the results of the engine.

It is worthwhile to note, at this point, the general nature of SAAPHIRE function calls. Almost every SAAPHIRE function returns an SI_Error structure as a result. This allows the calling function to determine the success or failure of the SAAPHIRE call and take appropriate action. Because of this protocol, most SAAPHIRE code looks like a tree's roots. There are "if" statements testing the success of a SAAPHIRE call and, if successful, more SAAPHIRE calls are made inside of the body of the "if" statement. This can make the code somewhat confusing and hard to follow at times. Adequate commenting eases the reader's eyes but such is the nature of SAAPHIRE programming (even with the included examples).

The init() function starts by initializing SAAPHIRE with a call to SAA_sceneInit(). Every SAAPHIRE effect begins with this call to initialize internal structures and fields. The init() function then allocates space for the InstanceData structure and initializes it with some necessary data such as a pointer to the scene, a pointer to a "custom value list" which contains the options selected by user on the dialog box, and the arguments picked by the user.

The GetDialogData() function is then called to initialize the fields of the InstanceData structure which hold the user's specified options. InitInstanceData() is called to perform more initialization. The now-initialized InstanceData structure is then attached to the custom context with a call to SAA_customContextSetInstanceData(). Now the InstanceData structure is ready to be accessed by the other entry point function(s).

Then the real magic occurs: a trueFlock object is instantiated and the flocking engine goes to work. The results are stored in the ActorPos array in the InstanceData structure.

Finally, to update the Softimage scene with the flocking engine's results, DoFCurves() is called.

cleanup (entry point function)

```
_CUS_EXTERN SI_Error cleanup (SAA_CustomContext context)
```

context – custom context for this effect

This is the cleanup entry point function named as the CLEANUP_FUNCTION in "trueFlock.cus". It is automatically called after the effect completes its execution. The cleanup() function frees all allocated memory. Specifically, the pointer fields of the InstanceData structure must be deallocated and the InstanceData itself must be freed.

This function also provides a convenient place for displaying debugging messages. For example, before deallocating the structure, the positions of the Actors can be printed to the standard output file to be examined.

GetDialogData (support function)

```
SI_Error GetDialogData (InstanceData * idata)
```

idata – pointer to InstanceData structure

This function retrieves the options set by the user on the dialog box and saves them in fields of the InstanceData structure.

The data is retrieved from a "custom value list" with functions of the form SAA_cusvalGet<Type>Value() where Type can be: "Int" for integer values, "Float" for floating point values, or "State" for Boolean values of radio-buttons and checkboxes.

The following describes the fields of the InstanceData structure which are affected and their purpose:

FrameStart (int) – starting frame number
FrameEnd (int) – ending frame number
NumFrame (int) – number of frames (FrameEnd – FrameStart + 1)
UniverseRadius (double) – radius of universal sphere
MaxAccel (double) – maximum acceleration per frame
MaxAngle (double) – maximum turning angle per frame
Priority (double) – array containing priorities of each instinct
FlockSphere (double) – radius of "nearby" flockmates sphere (for Flock instinct)
FlockCloseNum (int) – maximum number of "nearby" flockmates (for Flock instinct)
FlockDensity (double) – value indicating how far Actors should keep apart from each other
ActorVision (double) – range of Actor's vision (for Avoid Obstacles instinct)
FollowScript (int) – enable or disable FollowScript instinct

InitInstanceData (support function)

```
SI_Error InitInstanceData (InstanceData * idata)
```

idata – pointer to InstanceData structure

This function initializes the actor and obstacle arrays in the InstanceData structure. These arrays hold pointers to the models which represent the Actor and Obstacle objects in the scene.

If the user selected to randomly generate initial positions for the Actors, this function duplicates the picked model the necessary number of times.

This function also allocates the required memory for the ActorPos array which will hold the results of the flocking engine.

DoFCurves (support function)

```
SI_Error DoFCurves (InstanceData * idata)
```

idata – pointer to InstanceData structure

Once the flocking engine has completed execution and the ActorPos array in the InstanceData structure has been filled with the positions for each Actor, for each frame, the Softimage scene needs to be updated to reflect these positions.

There are many ways to animate the models in a scene. Two possible methods are: key-framing and attaching a model to a curve or path (usually a spline). Key-framing means that the explicit position (translation) and rotation of each model is set for a certain number of "key" frames and the animation software interpolates and "fills in" the in-between frames. The second method does almost exactly identically.

Using SAAPHIRE version 1.1, this project was forced to use the first method. This is not a major hindrance but it does make the code a little more lengthy and complicated. The programmer has to worry about function curves for translation and rotation for each model and where the key points on these function curves are.

SAAPHIRE version 1.7 provides functions which allow the programmer to create a spline from a number of points and then to pick that spline as a path for a model. This probably would have simplified things somewhat. Nevertheless, this project uses function curves to animate the Actors' models in the scene.

To animate the models, a function curve is created for each model for a particular attribute. For attributes such as translation (position) and rotation, three function curves are required- one for each of the three coordinate dimensions: x, y and z. In truth, a test is performed to see if a function curve already exists for the model. If not, then a function curve is created. Otherwise, the existing one is used.

Subsequently, a number of key points are created on each curve. The times when these key points occur must then be set. For this project, the position of each model is set during each frame. SAAPHIRE provides functions to convert from frames to time (seconds) as required by the various functions involved.

After the times of the key points have been set, their values can be set. The translation (positional) values are retrieved from the ActorPos array in the InstanceData structure.

The result is code containing three nested loops. The can be summarized as follows:

```
For each actor
        For each coordinate x, y and z
                For each frame
```

If only the positions of the models were set using function curves, the models would flock but would always be pointing in the same direction. For example, if one flocked a number of spaceships, the spaceships would move around but would always be pointing upwards. This is not very pleasing to the eye. Therefore, the models are rotated so that they point in the directions in which they travel. Using some mathematics, the models aim along the vector created from their position for the current frame to the following one. This yields much more "realistic" animations. Appendix B of this document details this process.

GetActorPos (support function)

```
float GetActorPos (InstanceData * idata, int Actor, int Coord,
                   int Frame)
```

idata – pointer to InstanceData structure
Actor – index into array of Actor positions
Coord – x, y or z
Frame – frame number

This function is called from the DoFCurves() function to retrieve the position of the given actor, for a given coordinate, for the given frame. This information is retrieved from the ActorPos array in the InstanceData structure.

GetAngle (support function)

```
float GetAngle (InstanceData * idata, int Actor, int Coord, int
                Frame)
```

idata – pointer to InstanceData structure
Actor – index into array of Actor positions
Coord – x, y or z
Frame – frame number

This function returns the angle the given coordinate axis (x, y or z) makes with the vector from the Actor's position for the current frame to its position for the next frame.

To aim the model in the right direction, it must be rotated around the x and y axes. Note that no z-rotation is necessary.

For an explanation of how the rotation angles for the models are calculated, please refer to Appendix B.

Some advice for future SAAPHIRE developers: When you hear a "ringy" sound and a message box appears with the words "Fatal error: Soft will die", you have probably committed one of the following sins, somewhere in your code:

- You forgot to free memory that you allocated
- You divided by zero
- You didn't test the SI_Error result returned by a SAAPHIRE call which failed

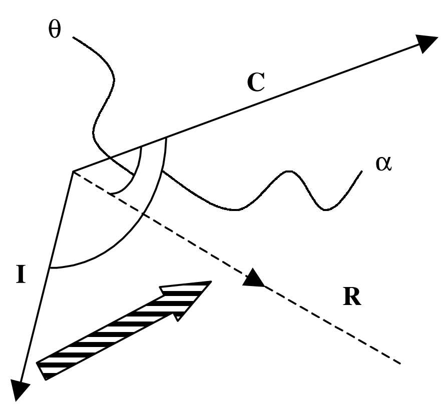
Appendix A

The following is a description of how the InstinctResult vector for each actor, during each frame, is rotated so that it does not exceed the maximum turning angle specified by the user. The vector is rotated such that two conditions are met:

1. The angle between the new InstinctResult and the Actor's previous course will equal the maximum turning angle.
2. The new InstinctResult will lie in the same plane formed by the old InstinctResult and previous course vectors.

The mathematics to achieve these results are quite involved. The Maple mathematics program was used to perform the initial derivations. The subsequent "peeling" to make the equations more easily adaptable to computer code was done by hand.

Let: **C** ≡ Actor's previous course
I ≡ old InstinctResult
R ≡ new InstinctResult
θ ≡ maximum turning angle
α ≡ angle between **I** and **C**



1. Condition #1 (part 1)

$$\cos(\theta) = \frac{\mathbf{C} \bullet \mathbf{R}}{|\mathbf{C}|\,|\mathbf{R}|}$$

Letting $|\mathbf{R}| = |\mathbf{I}|$,

$$\cos(\theta) = \frac{\mathbf{C} \bullet \mathbf{R}}{|\mathbf{C}|\,|\mathbf{I}|}$$

Rearranging,

$$\mathbf{C} \bullet \mathbf{R} = \cos(\theta)\,|\mathbf{C}|\,|\mathbf{I}|$$

$$\therefore c_x r_x + c_y r_y + c_z r_z = \cos(\theta)\,|\mathbf{C}|\,|\mathbf{I}| \qquad \mathbf{1}$$

2. Condition #1 (part 2)

Angle between **I** and **R**, $(\alpha - \theta)$, is given by:

$$\cos(\alpha - \theta) = \frac{\mathbf{I} \bullet \mathbf{R}}{|\mathbf{I}|\,|\mathbf{R}|}$$

Again, let $|\mathbf{R}| = |\mathbf{I}|$,

$$\cos(\alpha - \theta) = \frac{\mathbf{I} \bullet \mathbf{R}}{|\mathbf{I}|^2}$$

Rearranging,

$$I \bullet R = \cos(\alpha - \theta)\,|\mathbf{I}|^2$$

$$\therefore\ i_x r_x + i_y r_y + i_z r_z = \cos(\alpha - \theta)\,|\mathbf{I}|^2 \qquad \mathbf{2}$$

where α is the angle between **I** and **C**, given by:

$$\cos(\alpha) = \frac{\mathbf{I} \bullet \mathbf{C}}{|\mathbf{I}|\,|\mathbf{C}|}$$

$$\therefore\ \alpha = \cos^{-1}\left(\frac{\mathbf{I} \bullet \mathbf{C}}{|\mathbf{I}|\,|\mathbf{C}|}\right)$$

3. Condition #2

Normal to plane formed by **I** and **C** is given by:

$$\mathbf{N} = \mathbf{I} \times \mathbf{C} = \begin{cases} n_x = c_y i_z - c_z i_y \\ n_y = c_z i_x - c_x i_z \\ n_z = c_x i_y - c_y i_x \end{cases}$$

And, for **R** to lie in same plane, angle between **R** and **N** must be 90°

i.e.

$$\mathbf{R} \bullet \mathbf{N} = 0$$
$$n_x r_x + n_y r_y + n_z r_z = 0$$

$$\therefore\ (c_y i_z - c_z i_y) r_x + (c_z i_x - c_x i_z) r_y + (c_x i_y - c_y i_x) r_z = 0 \qquad \mathbf{3}$$

Now, from matrices A and B:

$$\mathbf{A} = \begin{bmatrix} c_x & c_y & c_z \\ i_x & i_y & i_z \\ c_y i_z - c_z i_y & c_z i_x - c_x i_z & c_x i_y - c_y i_x \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} \cos(\theta)\,|\mathbf{C}|\,|\mathbf{I}| \\ \cos(\alpha - \theta)\,|\mathbf{I}|^2 \\ 0 \end{bmatrix}$$

Solve the linear equation:

$$\mathbf{A}\ \mathbf{R} = \mathbf{B}$$

Maple gives a horrendous equation for R… This has been hand-“simplified” to:

$$r_x = \frac{-\cos(\alpha)\,|\mathbf{C}|\,|\mathbf{I}|\,[\,i_y\,(c_y\,i_x - c_x\,i_y) + i_z\,(c_z\,i_x - c_x\,i_z)\,] + \cos(\alpha - \theta)\,|\mathbf{I}|^2\,[\,-c_y\,(c_y\,i_x - c_x\,i_y) + c_z\,(c_z\,i_x - c_x\,i_z)\,]}{\%1}$$

$$r_y = \frac{\cos(\alpha)\,|\mathbf{C}|\,|\mathbf{I}|\,[\,i_x\,(c_y\,i_x - c_x\,i_y) + i_z\,(c_y\,i_z - c_z\,i_y)\,] - \cos(\alpha - \theta)\,|\mathbf{I}|^2\,[\,c_x\,(c_y\,i_x - c_x\,i_y) + c_z\,(c_y\,i_z - c_z\,i_y)\,]}{\%1}$$

$$r_z = \frac{\cos(\alpha)\,|\mathbf{C}|\,|\mathbf{I}|\,[\,i_x\,(c_z\,i_x - c_x\,i_z) - i_y\,(c_y\,i_z - c_z\,i_y)\,] + \cos(\alpha - \theta)\,|\mathbf{I}|^2\,[\,-c_x\,(c_z\,i_x - c_x\,i_z) + c_y\,(c_y\,i_z - c_z\,i_y)\,]}{\%1}$$

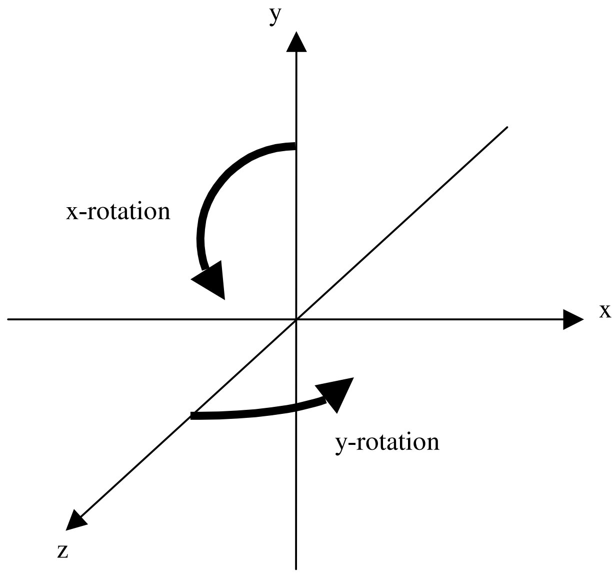
where

$$\%1 = -2\,(c_x\,c_y\,i_x\,i_y + c_x\,c_z\,i_x\,i_z + c_y\,c_z\,i_y\,i_z) + i_x^2\,(c_y^2 + c_z^2) + i_y^2\,(c_x^2 + c_z^2) + i_z^2\,(c_x^2 + c_y^2)$$

These equations were transformed into “as-efficient-as-possible” C++ code and placed in the FixVector() method of the FlockEngine class.

Appendix B

The following is a description of how the rotation angles are calculated in order to point the Actor models in the directions they are travelling during each frame.



The x-rotation angle refers to the rotation around the x-axis starting from the positive y-axis in a counter-clockwise direction towards the positive z-axis.

The y-rotation angle refers to the rotation around the y-axis starting from the positive z-axis in a counter-clockwise direction towards the positive x-axis.

To rotate a model, the model is rotated around the x-axis through an angle between 0 and 180 degrees. This "tilts" the model downwards. The model is then rotated around the y-axis with an angle between 0 and 360 degrees. This "turns" the model in the direction it is travelling. No rotations around the z-axis are required.

First, we deal with the y-rotation angle. Note that when rotating about the y-axis the y component of the vector is irrelevant. Therefore, we can work in the xz-plane. The y-rotation angle, θ, then, is the angle the vector $\mathbf{V} = (v_x, v_y, v_z)$ makes with the positive z-axis, $\mathbf{k} = (0, 0, 1)$.

Using the equation for the angle between two vectors:

$$\cos(\theta) = \frac{\mathbf{k} \bullet \mathbf{V}}{|\mathbf{k}|\,|\mathbf{V}|}$$

Simplifying, we get:

$$\cos(\theta) = \frac{v_z}{\mathrm{sqrt}(\ v_x^{2} + v_z^{2}\ )} \qquad \mathbf{1}$$

Because the y-rotation can be an angle between 0 and 360 degrees, we must check to see if the angle is greater than 180°. That is, if $v_x < 0$. If so, we modify the y-rotation angle to be $\theta' = 360° - \theta$.

Now the method of calculating the x-rotation angle, ϕ, is given. As above, when working with a rotation around the x-axis, the x component of any vector becomes irrelevant. Therefore, we work in the yz-plane. We find the angle between the positive y-axis, $\mathbf{j}$ = (0, 1, 0) and a modified version of the previous vector, **V'**. This vector is the vector **V** which has been rotated around the y-axis to lie in the yz-plane. Because the magnitude of a vector is not altered when it is rotated (rotation is a non-affine transformation), this modified vector, **V'**, will have the same magnitude as **V**. Therefore, **V'** = $(0, v_y, v'_z)$.

Now,

$$\begin{aligned} |\mathbf{V}| &= |\mathbf{V'}| \\ \text{sqrt}(\, v_x^2 + v_y^2 + v_z^2\,) &= \text{sqrt}(\, v_y^2 + v'^2_z\,) \\ v_x^2 + v_y^2 + v_z^2 &= v_y^2 + v'^2_z \\ v'^2_z &= v_x^2 + v_z^2 \\ v'_z &= \text{sqrt}(v_x^2 + v_z^2\,) \end{aligned}$$

Therefore, we must find the angle between $\mathbf{j}$ = (0, 1, 0) and **V'** = $(0, v_y, \text{sqrt}(v_x^2 + v_z^2\,)\,)$.

This angle is given by:

$$\cos(\phi) = \frac{\mathbf{j} \bullet \mathbf{V'}}{|\mathbf{j}|\,|\mathbf{V'}|}$$

Simplifying,

$$\cos(\phi) = \frac{v_y}{\text{sqrt}(\, v_y^2 + v_x^2 + v_z^2\,)}$$

$$\cos(\phi) = \frac{v_y}{|\mathbf{V}|} \qquad \textbf{2}$$

Because the x-rotation angle is between 0 and 180 degrees, no extra work is necessary.

Using the above two equations, we can calculate the x and y rotation angles for each Actor model, for each frame of the animation.