

CIS 462/CIS562 Computer Animation (Fall 2017) Homework Assignment 8

(Particle Systems)

Due: Tuesday, Nov. 30, 2017 (must be submitted before midnight)

Please note the following:

- No late submissions or extensions for this homework.
- This is a programming assignment. You will need to use the *AnimationToolkit* code framework from the previous CIS462/562 IK Player assignment
- Unzip the HW8-AnimationToolkit.rar file and copy the appropriate files into the AnimationToolkit folder on your local machine.
- Double click on the file “Demo-ParticleViewer.exe” in the AnimationToolkit\bin directory to see a demo of some of the basic particle system functionality you will need to implement in this assignment. Use the corresponding ParticleViewer project to debug the functions you will need to implement in the APartile.cpp class. Double click on the file “Demo-FireworksViewer.exe” in the AnimationToolkit\bin directory to see a demo of fireworks simulation you will be implementing.
- When you are finished with this assignment, Commit and push the updated project files to your CIS462-562 GitHub repository.
- Work within the *AnimationToolkit* code framework provided. Feel free to enhance the AntTweakBar GUI interface if you desire to add more buttons or system variables
- You only need to implement the functions in the *.cpp files marked with “TODO” to complete this assignment.
- **NOTE: THIS IS AN INDIVIDUAL, NOT A GROUP ASSIGNMENT.** That means all code written by you for this assignment should be original! Although you are permitted to consult with each other while working on this assignment, code that is substantially the same as that submitted by another student will be considered cheating.

ANIMATION TOOLKIT – PARTICLE SYTEMS

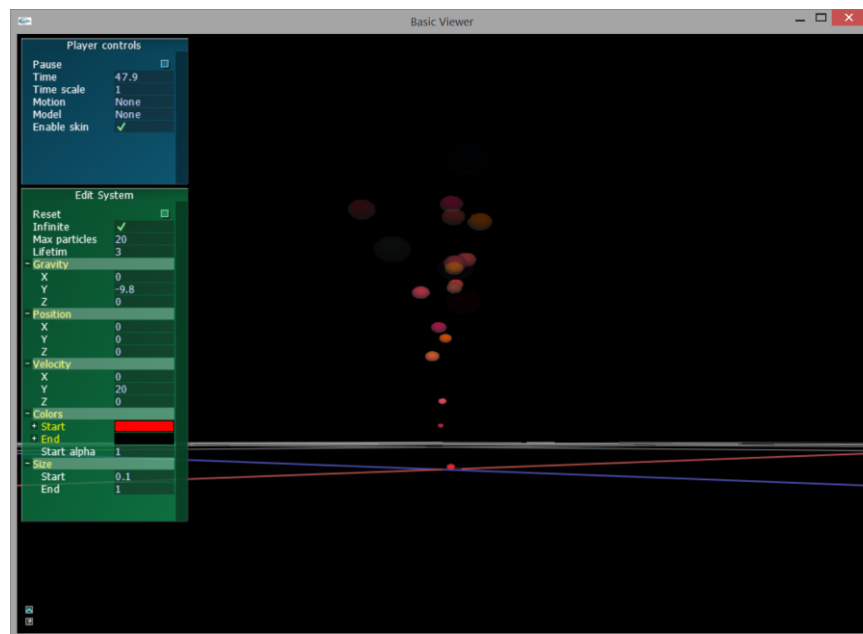
The goal of this assignment is for you to become familiar with the implementation of particle system dynamics and physical simulation. This assignment has three parts:

Part I – Basic C++ Particle System Simulation (20 points)

In this part of the assignment you need to implement a basic particle system simulation. The framework provided contains a particle system class whose job is to emit particles. Each particle then has a set of properties -- e.g. position, orientation, color number -- which changes over time.

About the ParticleViewer Project

This assignment builds on the previous assignments, adding a new library and project applications. If you did not complete a previous assignment, you may edit the basecode project files to use the corresponding solution library. To do this, right click on the project, select Properties, and then expand the Linker options. Select 'Input' and then edit the 'Additional Libraries' to point to libAssignmentX-soln.lib instead of libAssignmentX.lib. You will need to do this for both the release and debug versions of your project.



The basecode includes a simple interface and 3D viewer which creates a particle system located at (0, 0, 0). There is a UI for setting each particle system option which you can use for testing.

The camera control in the ParticleViewer is the same as in the BVHViewer:

- Left-button drag with the mouse to rotate
- Right-button drag with the mouse to pan
- Middle-button drag with the mouse to zoom
- 'f' button will focus the view on the character

Assignment Details

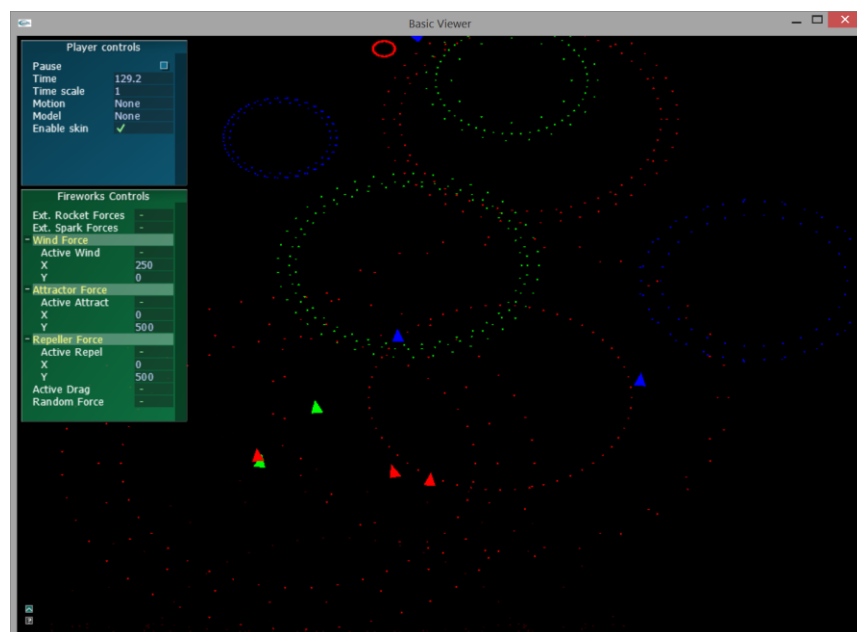
1. (20 points) Implement the various functions in the AParticle.cpp file indicated by “TODO: Insert your code here”
 2. (up to 10 points extra credit) Integrate billboards with transparency into the framework to create different effects such as smoke, fire, or butterflies (be creative!)
-

Part II – C++ based Fireworks Simulation (55 points)

In this part of the assignment you will use the particle class implementation developed in part 1 to create a fireworks simulation. When you press the space key, a rocket will fire into the air with a randomized velocity and time to live. When the rocket's time to live expires, it will explode into concentric rings of sparks. When the sparks reach the ground (at $Y=0$), they should bounce. Both the rocket and the sparks are represented by particles..

About the FireworksViewer Project

The basecode includes a simple 2D viewer for launching rockets. To launch a rocket, press the space key. For the purposes of the framework, particle movement is restricted to the X-Y plane.



Assignment Details

1. (55 points) Make some fireworks. Implement the various functions in the ASpark.cpp and AFireworks.cpp files indicated by “TODO: Insert your code here”
2. (15 points) Extra credit.
 - (10 points) Create a rocket trail effect by emitting spark particles based on the rocket position
 - (5 points) Extend the fireworks to 3D

Part III – Houdini- based Fireworks Simulation (25 points)

In the third part of this assignment your task will be to create the firework simulation using the Houdini FX software. Houdini is one of the most popular special effects software used in the VFX industry. The main advantage of Houdini is that it is node based and procedural, which means that you can change simulation parameters at any point in the work flow and get desired results instantly.

In this part of the assignment you need to do the following:

1. Download and install the Houdini Apprentice software from http://www.sidefx.com/index.php?option=com_content&task=blogcategory&id=245&Itemid=400 to create the fireworks simulation.

2. Check out the following links for info about Houdini and its user interface:

Houdini:

<http://tangdonna.blogspot.com/2011/04/sops-dops-pops-rops-vops-shops-and.html>

User Interface:

http://www.sidefx.com/index.php?option=com_content&task=view&id=1869&Itemid=347

3. Although Houdini has a default option to create a fireworks simulation, in this assignment you will need to create the whole fireworks simulation from scratch using the following nodes and networks since they are the most important components when building a dynamic simulation:

Nodes: Split nodes, Group nodes, and Collect nodes

Networks: Geometry network, Particle operator network and Shader operator network.

4. You then need to render the fireworks simulation using the Mantra renderer to create a video demo.

Extra credit. (10 pts per effect). In a manner similar to the C++ fireworks simulation create a spark effect which has particles being emitted when the falling particles collide with the ground. The important nodes to use here are event nodes and collision nodes (which will also help you understand how expressions are used in Houdini). Other effects to be implemented include the creation of different types of “firework patterns” after the rocket explodes through use of wind, attractive forces, repulsive forces, force fields, etc.

5. Please submit your Houdini fireworks project along with the video to the CIS462/562 Canvas site. Before you submit your assignment to Canvas, please put your Houdini project files into a zip file with the name “HW8_LastnameFirstnameHoudini.zip”. Also include in this zip file the associated video in either the MOV, MPEG4, or WMV formats (no AVI files PLEASE!) Finally include a description of each of the nodes you used in your simulation and explain the difference between the fireworks simulation with and without the event nodes.

Base Code Details:

Particle System Project Implementation Overview

The class AParticleSystem manages a set of particles relative to a root transform.

AParticleSystem also initializes default starting values for each particle. When AParticleSystem creates or emits a particle, it initializes the particle's values using these defaults.

- Particle systems can be infinite, meaning they continuously emit particles (e.g. for smoke), or finite, meaning they only emit a given of particles (e.g. for fireworks). In the next assignment, you will use the particle system created here to create fireworks effects.
- To keep memory needs constant, the particle system has a max number of particles. Particles are "recycled" when one particle dies and it is time to emit a new particle.
- The particle system manages external forces which are applied to its particles each frame. The default framework in the ParticleViewer project supports gravity only.
- The particles in the particle system can be configured to "jitter" values so that particles do not all look identical.

The class AParticle contains a set of properties which can change over time. Some of the important variables include:

- lifespan: how long in seconds a particle will live. For an infinite particle system, a particle is reborn after it dies.
- timeToLive: the amount of time left before the particle dies
- state: a vector (i.e. a 1D array) containing the state of the particle in terms of its position, velocity, forces, mass and time to live.
- stateDot: a vector (i.e. a 1D array) containing derivatives of the state vector
- Pos: the position of the particle
- Vel: the velocity of particle
- Color: the current color of the particle. The color can change over the lifetime of the particle based on the start/end values
- Scale: the current color of the particle. The scale can change over the lifetime of the particle based on the start/end values
- Alpha: the current transparency of the particle. The transparency can change over the lifetime of the particle based on the start/end value
- mass: the mass of the particle

Particle position (**p**) and velocities (**v**) are updated each frame based on the forces applied using either Euler or Runge Kutta2 integration. For example, with Euler integration:

$$\begin{aligned}\mathbf{p}(t_{k+1}) &= \mathbf{p}(t_k) + \Delta t \cdot \mathbf{v}(t_k) \\ \mathbf{v}(t_{k+1}) &= \mathbf{v}(t_k) + \Delta t \cdot \mathbf{a}(t_k)\end{aligned}\quad \text{where } \mathbf{a}(t_k) = \text{force}(t_k) / m$$

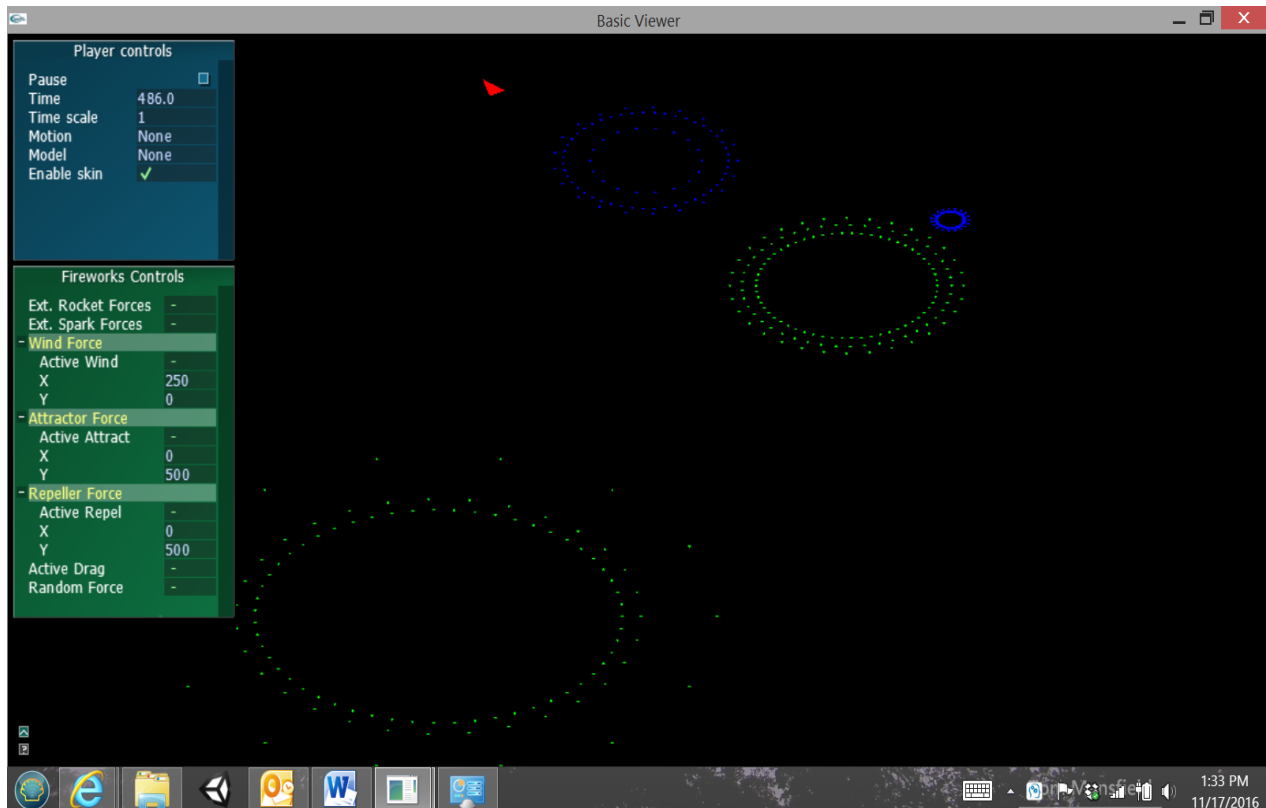
Fireworks Project Implementation Overview

AFireworks simulates rockets and sparks in the simulation through calls to the *AFireworks::update()* function.

At first, there are no particles, so nothing is displayed. When the user hits the space key the function *AFireworks::fireRocket()* function generates a rocket with random velocity (in 2D). As the simulation continues (the user can also fire more rockets). Once launched, the rocket will reach its maximum height (due to gravity) and explode.

From this point in the simulation onward, for *TOTALEXPLOSION* time steps, *AFireworks::explode()* function will generate 5 rings of sparks evenly distributed. The number of sparks per ring and the explosion velocity are random numbers (see comments in *AFireworks* cpp files for details). Sparks are also simulated in the system and have a fixed life time. They might contact the ground, and bounce off with a given coefficient of restitution.

Here's a screen shot of the application:



Important Class Details

AParticle is the base class of all particles. Every particle (either a rocket or a spark) inherits from this base class.

Relevant Member variables:

dim: Dimension of the state vector. By default $\text{dim} = 12$;

state: State vector.

- * 0 : position x
- * 1 : position y
- * 2 : position z
- * 3 : velocity x
- * 4 : velocity y
- * 5 : velocity z
- * 6 : force x
- * 7 : force y
- * 8 : force z
- * 9 : mass
- * 10 : timeToLive
- * 11 : not defined

stateDot: Derivative of the state vector.

deltaT: Delta time of one simulation step

alive: Indicates if the particle is dead or not. If it is dead, it should not be displayed and should not be continued to be simulated.

Relevant Member functions:

GetState(): Gets the state vector. Returns the pointer to the state vector

SetState(newState): sets the state vector with value *newState*. Therefore, *newState* and state vector must have the same dimension.

Virtual functions are implemented by each sub-class that inherits from the *AParticle* class. Some of the important ones include:

ComputeForce(): Computes the forces to be applied to the particle.

ComputeDynamics(): Compute the derivative of the state vector (i.e. *stateDot*)

UpdateState(): Update the state vector with the derivative vector and perform other necessary calculations

update(deltaT): Computes one simulation step using either Euler or RK2 integration.

ASpark is a sub-class inherited from the *AParticle* class. It inherits everything (member variables and functions) from *CParticle* and has its own properties.

Relevant Member variables:

color[3]: spark color. It is set by the rocket that it is emitted from.

COR: Coefficient of restitution. Determines how much the spark bounces when it hits the ground.

Member functions you need to implement. See cpp file for details.

computeForces(): Computes the forces acting on the particle. These include gravity plus any other forces such as wind, attraction, repulsion, random, etc.

resolveCollisions(): Computes changes to the velocity vector when the particle hits the ground based on the coefficient of restitution (COR)

ARocket is a sub-class derived from the *ASpark* class. It inherits everything (member variables and functions) from *ASpark* and has its own properties.

Relevant Member variables:

explosionCount: Time to go of explosion phase. It should count down from TOTALEXPLOSION to 0 during a explosion.

mode: Current mode of the rocket, see enum ROCKETMODE {FLYING, EXPLOSION, DEAD}.

Vexplode: min vertical velocity for rocket to explode

AFireworks creates and simulates the rockets and sparks.

Relevant Member variables:

vector<CRocket> rockets*: STL Vector of pointers to *ARocket*. Rockets in this vector can either be flying or generating sparks

vector<CSpark> sparks*: STL Vector of pointers to *CSpark*. Sparks that are generated by the exploding rockets.

rocketMass: mass for rocket (default is 50).

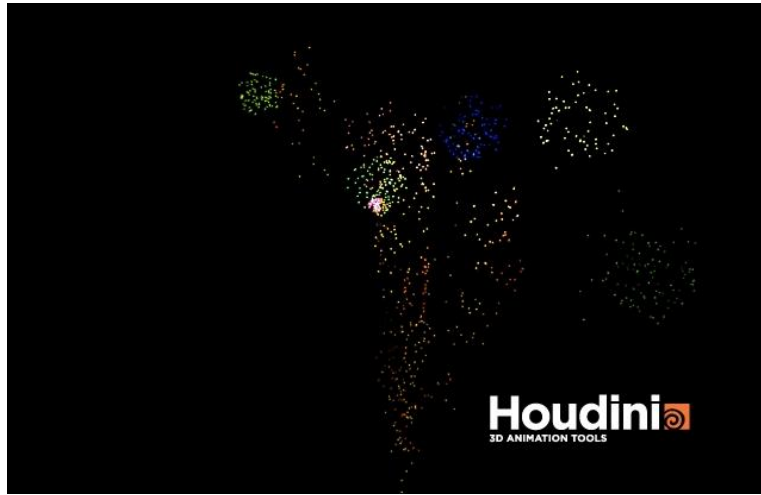
sparkMass: mass for rocket (default is 1).

Functions that you need to implement. cpp file for details.

void explode(): When a rocket reaches its maximum height, it explodes and calls this function to generate sparks

void fireRocket(): When a key is pressed, this function is called to generate a rocket.

Part III – Houdini- based Fireworks Simulation



Step 0:

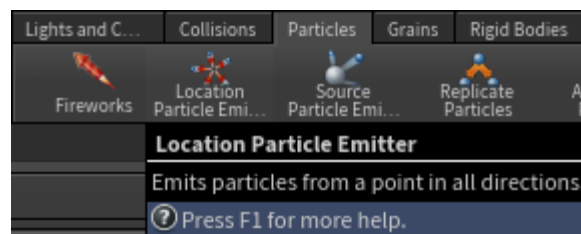
Go to <http://www.sidefx.com/> and download the free Apprentice version that is available. The Apprentice version will insert a small watermark in your renders and limit the render size, but that is fine for this assignment. Houdini Apprentice is available for Windows, Mac, and Linux. This tutorial was written for version 16.0.600.

Step 1:

Familiarize yourself with the UI. Houdini has several quick start tutorials on their website [here](#). You should learn to navigate the scene, use the parameter and network views, and create nodes.

Step 2:

Let's create a particle emitter for the firework projectiles. Open the Particles shelf and click Location Particle Emitter. Hover your mouse over the viewport and press Enter to accept the default placement at the origin.



Step 3:

Press play! By default, Houdini will play as fast as it can. To change playback to real-time, click on the clock at the bottom right:



Notice that the timeline will turn blue to indicate that the frames have been cached. If it's orange, it's because you changed the simulation partway through an existing simulation, so you should restart your playback from the first frame.



Step 4:

We should now be inside the AutoDopNetwork in the network view. Take a look at the nodes. Middle-mouse-click on any of the nodes to learn more.

- The popobject1 node represents the object we see in the viewport.
- The merge nodes simply merge multiple streams of particles.
- The popsolver1 node solves the forces on the particles and updates the particles' attributes.
- The gravity1 node applies gravity to all our objects.
- The location node generates the initial values of all the particles.

Step 5:

First, let's create the shell particles using the location node. These will represent the packaged fireworks *before* they explode into colorful sparks. Inside the parameter view, we'll need to change some parameters for the particles to be more like the shells.

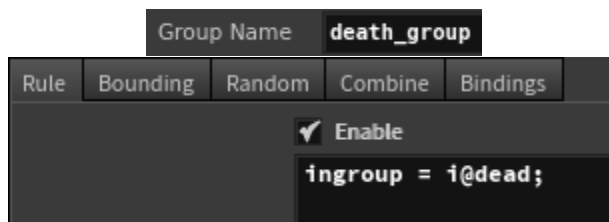
- Hover over the location node's attributes in Houdini to learn more.
- Birth >> Constant Birth Rate – Lower this so that we only have several fireworks at once.
- Birth >> Life Expectancy – This represents the time until the particle dies. We'll use it so that the fireworks explode when this happens. Adjust accordingly.
- Birth >> Life Variance – Add some randomness to make the simulation more realistic.
- Attributes >> Velocity – This represents the initial velocity of the particles. Change this so that particles are shot up into the air.
- Attributes >> Variance – Change this to add variety to the initial velocities of the particles.

With enough tweaking, you should have a few shell particles shooting up every second and dying when they reach near their apex in the air.

Step 6:

Now we want to have the firework shell explode into many more spark particles when it dies. To do this, we can use particle groups, which allow us to reuse these dead particles elsewhere.

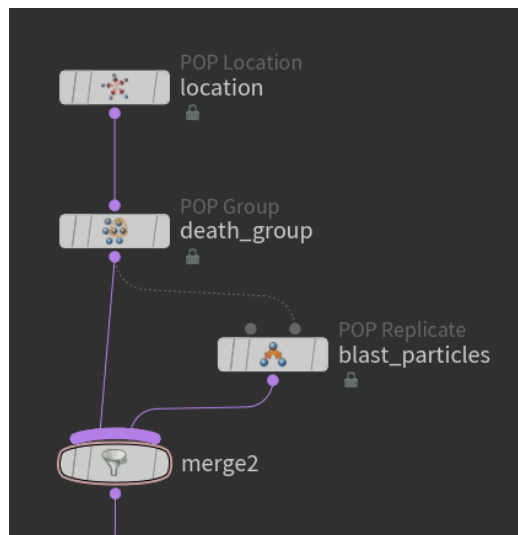
We'll create a group of the dead particles that we can use to spawn additional particles. Add a POP Group node in between the location node and the merge2 node. In the parameter view, give the group a name like "death_group". Then, enable the Rule tab so that we can filter out the particles using VEX, Houdini's own special language for running custom code. Learn more [here](#).



Set the rule to `ingroup = i@dead;` Notice the semicolon. This is roughly equivalent to `bool ingroup = isdead;` The expression is tested on each of the particles from the location node, and if `ingroup` is true, the particle is added to the “death group”. Rename the POP Group node to “death_group” so that it’s easy to remember the group name.

Step 7:

Now we want to duplicate each dead firework shell into many, many spark particles. Create a POP Replicate node. We’ll rename it to `blast_particles`. Create a connection from the output of `death_group` to the Reference Stream input of `blast_particles`. This essentially creates a separate particle stream that we can use for blast particles only. Then connect the output of `blast_particles` to `merge2`. Remember you can hover over an input to see what it’s for.



We only want to duplicate `death_group` particles here, which represent firework shells that are ready to explode. So, check the Group checkbox in `blast_particles`’s parameter view, then set Group to “death_group”. If you play now, you’ll notice that the explosion is pretty dull.



Step 8:

Before we change the settings of `blast_particles`, we want to apply an air drag force to the particles for realism, which can affect our parameters. Add a POP Drag node in between `blast_particles` and the `merge2` node.

Now we can change some parameters in `blast_particles`:

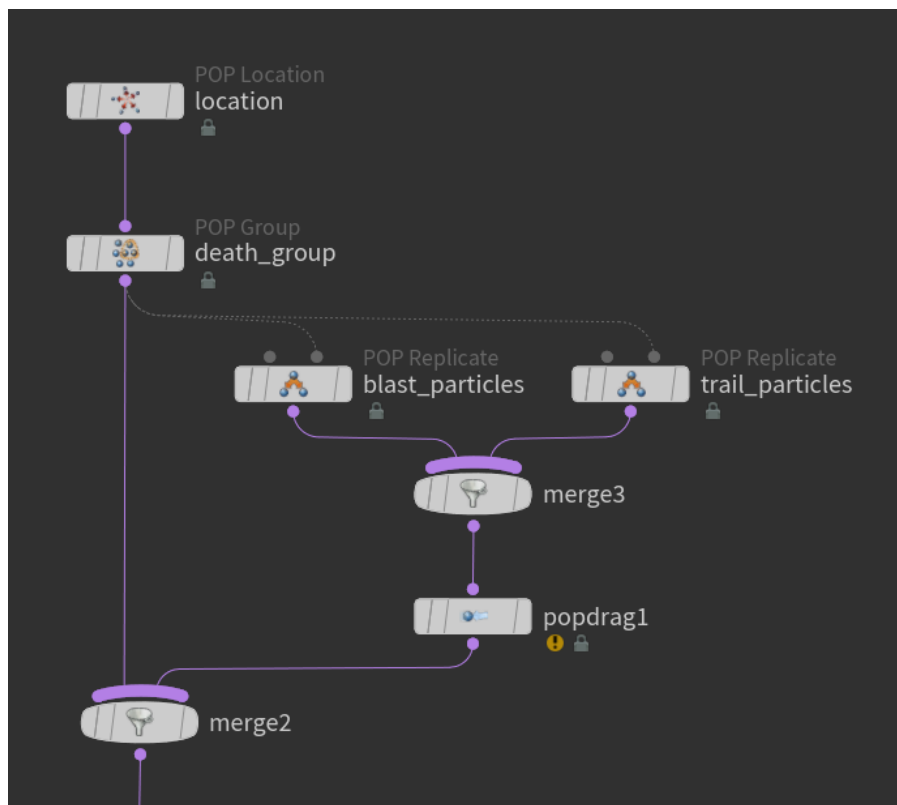
- Birth >> Constant Birth Rate – Set this to 0. We’ll use impulse activation here since particles only have one frame to spawn.
- Birth >> Impulse Count – Adjust to your liking. This is the number of particles spawned each time the replicate node is run on a particle.
- Birth >> Life Expectancy – Set the lifetime of the sparks to something realistic.
- Birth >> Life Variance – Add variance to the lifetime for more realism.
- Shape >> Shape – Change to “Point” so that the sparks spawn from the original point’s location.

- Attributes >> Initial Velocity – Change to “Add to inherited velocity” to mimic an additive explosive force since sparks inherit their shells’ velocities.
- Attributes >> Variance – This represents the scale of the explosion, the randomness of the direction and magnitude of each spark particle. Adjust to your liking.
- Feel free to adjust the amount of air resistance in the drag node.

Step 9:

Let’s add a trail of smoke (or spark) particles to the firework shell. Add another POP Replicate node named “trail_particles” and connect the output of death_group to the Reference Stream input of trail_particles. Don’t change the group settings for this node though! We always want the shell particles to emit a trail, not just when they die.

We also want to apply our drag force node to both blast_particles and trail_particles, but popdrag1 will only accept one particle stream at a time. To work around this, create a new Merge node and connect blast_particles and trail_particles to this node. Now, we can connect the merge node to the drag node.

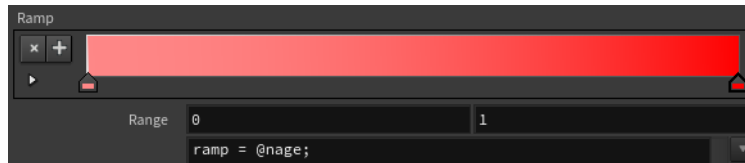


We could have connected trail_particles to the location node instead of the death_group node and gotten the same results. The only difference would be that trail_particles wouldn’t know about the death_group.

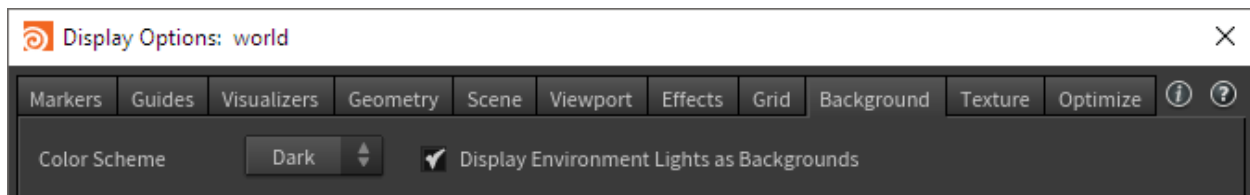
You should now be able to adjust the familiar settings in trail_particles to achieve the trail effect that you want. Lower the value of Attributes >> Inherit Velocity to mimic expulsion forces from the firework. This is because the expulsion forces on the trail particles negate the inherited velocity from the shell particles.

Step 10:

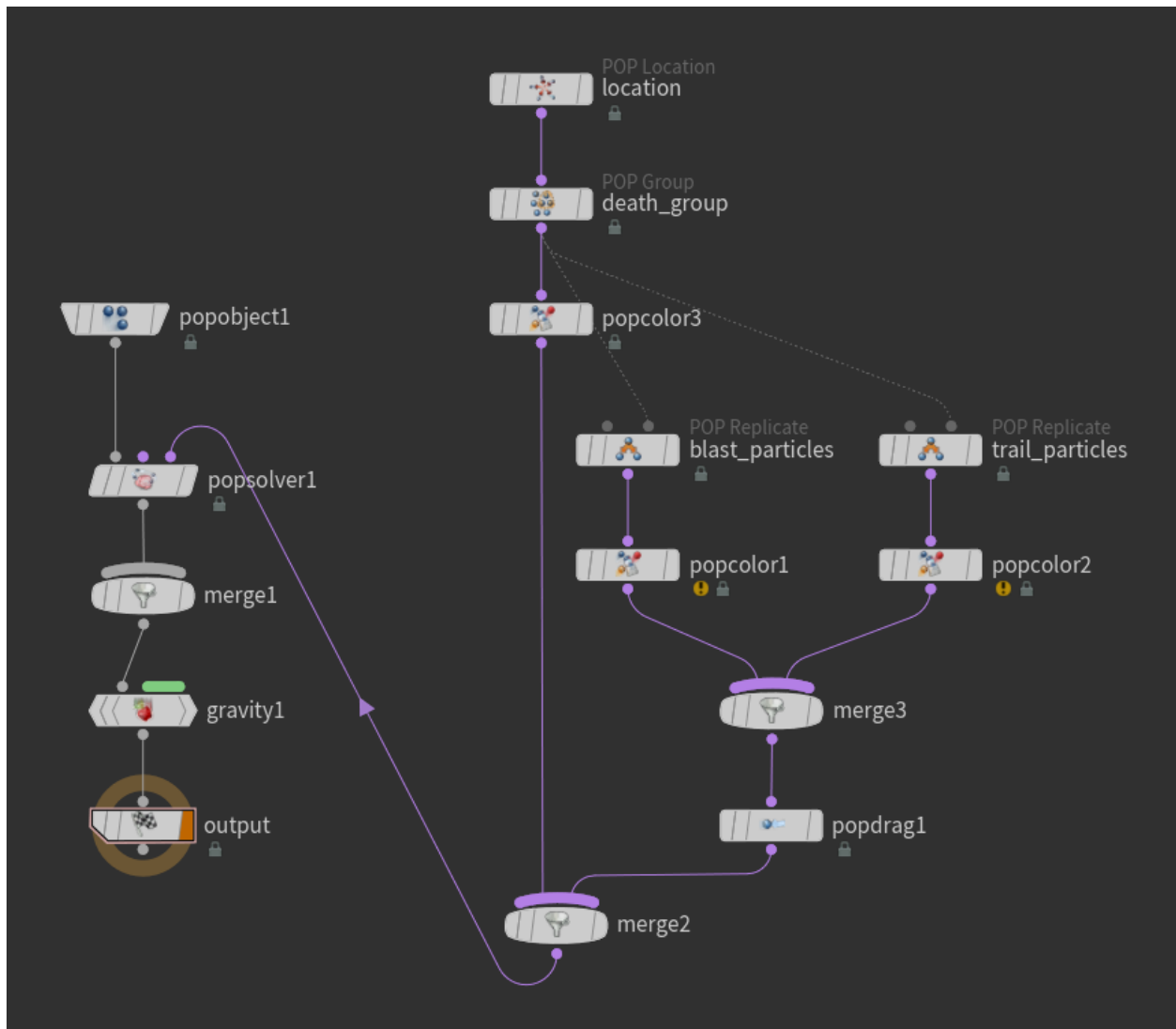
Your fireworks should look much better now! Add a POP Color node to each of your three particle streams to give the particles some vibrant color. You can set Color >> Color Type to Ramp to change the color of the particle over its lifetime. Notice that the VEXpression is set to “`ramp = @nage;`” to achieve this. @nage is the normalized age from 0 to 1. Play around with other color types and VEXpressions!



The default blue background makes it difficult to judge the colors. You can change the viewport color by hovering your mouse over the viewport and pressing D. This should open the Display Options. Change Background >> Color Scheme to Dark.




You're finished! You should now have a realistic firework simulation. The resulting AutoDopNetwork should look something like this:



Step 11:

The final task is to render the simulation. Follow these steps:

- Right-click on the flipbook on the bottom left of the viewport: 
- Click “Flipbook with New Settings...”
- Change the size of the render to be less than or equal to 1280x720, which is the maximum allowed render resolution for Houdini Apprentice.
- Click “Accept”. This will render what you see in the viewport into Houdini’s MPlay player, much like Maya’s Playblast feature.
- If you are on Mac:
 - File >> Export >> QuickTime Movie Exporter
 - Save and rename the resulting file to HW8_LastnameFirstname.mov
- If you are on Windows or Linux:
 - Exporting to video doesn’t seem to work with Houdini Apprentice.
 - Click “File >> Save Sequence As...”

- Change the export filename to something like `imageName$F.tga`. `$F` is the VEX variable for the frame number. Targa files work well with most video encoders.
- Click “Save”. This will save the rendered image of each frame of your simulation.
- Use your favorite video editor to convert the image sequence. If you don’t have one, download and install [FFmpeg](#), an open-source video encoder.
- Run the command: `ffmpeg -r 24 -f image2 -i imageName%d.tga -vcodec libx264 HW8_LastnameFirstname.mp4`

Congratulations on your work with Houdini!

Troubleshooting

- If Houdini stops responding
 - You may have too many particles. Lower your birth rates.
- If your simulation is missing some parts or is not appearing at all
 - Make sure that the output node has the output flag set



- If on Mac, your flipbook isn’t rendering
 - You may need to allow Houdini through your Mac’s firewall so that Houdini can send itself its renderings.
 - In System Preferences >> Security & Privacy >> Firewall >> Firewall Options...
 - Then allow incoming connections for Houdini FX and Image Viewer.
- If your POP Color nodes have “Invalid Group” warnings
 - This is usually normal because there aren’t any particles to make groups from. However, if it persists, continue reading.
 - Check for any typos or invisible characters in “death_group”.

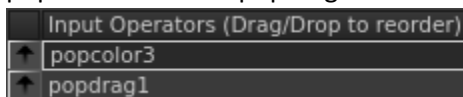
- Make sure this is inside of the POP Group node's "Group Name" attribute.

Group Name **death_group**

- Make sure this is inside blast_particle's "Group" attribute with the checkbox next to it enabled.

☒ Group **death_group**

- In the merge node closest to the POP Solver, make sure that the stream coming from the `death_group` node is first.
 - In our example network, if you click on the merge2 node, you should see that `popcolor3` is above `popdrag1`. Reorder them to get something like this:



- The rationale is that the primary stream (i.e.: the location stream) should run first so that any particles that die will be kept for the reference streams (i.e.: the trail and the blast particles).