EECS 351-2: Intermediate Computer Graphics
Instructor: Jack Tumblin

ProjectA: Particle Systems

Spring 2018

Project Demo Day (and due date): Tues May 8, 2018

Make an animated, interactive program WebGL that makes amazing and delightful 3D pictures from multiple parallel independent systems of evolving particles. We explored a wide range of choices for forces, constraints, solvers and independent and coupled particle behaviors: implement those you find most intriguing and inventive.

Requirements:

A)--In-Class Demo Day: when you demonstrate your completed program to the class. Two other students each evaluate your work on a 'Grading Sheet', as may Tumblin and assistants. Based on Demo Day advice, you then have ≥72 hours to revise and improve your project before submitting the final version for grading. Your grade will mix Demo Day grading sheets + your improvements.

B)--Submit final version on CANVAS no more than 72 hours +1 weekend-day later (Saturday May 12, 11:59PM) to avoid late penalties. Submit just one single compressed folder (ZIP file) that contains:
1) your written project report as a PDF file, and
2) one folder that holds sub-folders with all Javascript source code, libraries, HTML, etc. (mimic the 'starter code' ZIP-file organization). We must be able to read your report & run your program in the Chrome browser by simply uncompressing your ZIP file, and double-clicking an HTML file found inside, in the same directory as your project report.
---**IMPORTANT:** Name your ZIP file and the directory inside as: **Familyname**Personalname_ProjA
For example, my project A file would be: TumblinJack_ProjA.zip. It would contain sub-directories such as 'lib' and files such as TumblinJack_ProjC.pdf (a report), TumblinJack_ProjA.html, TumblinJack_ProjA.js ,etc.
---To submit your work, upload your ZIP file to Canvas→Assignments. DO NOT e-mail your project (not accepted!).
---BEWARE! SEVERE LATE PENALTIES! (see Canvas→Assignments, or the Syllabus/Schedule).

Project A consists of:

1)—Report: A short written, illustrated report, submitted as a printable PDF file.
Length: >1 page, and typically <6 pages, but you should decide how much is sufficient.
A complete report consists of these 4 sections:

a)--your name, netID (3 letters, 3 numbers: my netID is jet861), and a descriptive title for your project
(*e.g.* "Project A: How Tornadoes disrupt Flags, Jello, Fireworks and Birds," not just "Project A")

b)—a brief 'User's Guide' that explains your goals, and includes complete instructions on how to run and control the program (*e.g.* "A,a,F,f keys aim the fire-hose-like spray of particles, S,s,D,d keys drop the cannon-balls onto the trampoline; arrow UP/DN keys speed up/slow down the tornado, text below the tornado shows velocity of the fastest-moving particle within it).
CAUTION: Your classmates should be able run, understand, and extend your project from this report alone!

c)—a brief 'Code Guide' that explains your classes/data structures/files, and also how they're accessed and used. You can refer readers to well-commented code rather than re-explaining it, but explain enough for your work to be sensible and extensible by your classmates.

d)— a brief, illustrated 'Results' section that shows **at least 5 still pictures** of your program in action (use screen captures; no need for video capture), with figure captions and text explanations.

2)--Your Complete WebGL Program, which must include:

a)--On-Screen User Instructions: Your program must always easily explain itself; it should never puzzle its users. Users must be able to find help easily, without having to read your report or ask you for help. How? You decide! Perhaps you could put instructions on the webpage outside the 'canvas' element, or in a pop-up CSS window, or add a 'HELP' button, or use the WebGL book's 'HUD' method (Matsuda book, pg.368), or on-screen text: 'press F1 (Help) key to print instructions on console', *etc*. Onscreen it's best to give brief but complete, specific instructions; *e.g.* 'press arrow keys to move skateboard, drag left mouse key to steer through maze, right mouse drag to aim the flame-thrower'.

b)—**3D Interactive Viewing**: users must be able to move & aim the 3D camera in 3D. You must demonstrate how to move your camera to any desired 3D position independently by moving in any desired direction (e.g. aim camera in one direction and move in that direction; apply the 'glass tube' analogy), and demonstrate how to change the camera's aiming direction independently without changing its 3D position. You will need this and possibly more to allow graceful fly-throughs, or even make the camera follow a 'flock' of particles.

c)—Vast 'Ground Plane' Grid: Your program must clearly depict a horizontal 'ground plane' that extends to the horizon: a very large, repetitious pattern of repeated lines, triangles, or any other shape that repeats to form a vast, flat, fixed 'floor' of your 3D world. You MUST position your grid **in the x,y plane** (z=0) of your 'world-space' coordinate system. **(DO NOT create your ground plane in the world-space x,z plane!)** This grid should make any-and-all camera movements obvious on-screen, and form a reliable 'horizon line' when viewed with a perspective camera. HINT: In the camera or 'eye' coordinate system, the x,y plane is vertical, aligned with 2D image x,y. In 'world' coordinate system, the x,y plane is horizontal: z axis points up to the sky. I strongly recommend you use our textbook's gl.setLookAt()' function.

d)—**Generic 3D Particle System object or prototype** using state-variable description of the particles. Create one and only one generic particle system prototype object that, by changing nothing more than the values of its member variables, creates and depicts each and every possible kind of particle system we studied. Note items **f)** below: you will run at least 4 of these different particle systems simultaneously on-screen, yet each is an instance of exactly the same CPartSys class with initialized differently.

STATE VECTORS REQUIRED: This generic 'CPartSys' object must contain state vectors; unified, vector-like data types that hold all particles of one particle system, and have simple vector-like member functions/operators—add, subtract, scale; that work equally well for all possible kinds of particle systems. You'll need to compute with state vectors s0, s1, s0dot, and probably more (sM, sMdot, s2, s2dot, etc) by vector math: sM = s1 +(h/2)*s0dot; etc. **State vectors are essential for writing different kinds of solver functions;** without them, you must write a different solver for each of your 4 systems! Attempts to avoid using state variables, using their contents as separate variables or arrays just won't work here (e.g. "My 'state variables' are in my head: I know them well as these 12 separate variables and arrays for mass, position, velocity, forces, etc." this will not suffice), because your solvers will grow far too rapidly into an overly-elaborate, repetitive, and bug-prone mess, with little or no flexibility/adaptability to later, more sophisticated particle systems. Don't do it!

e)—This one master particle-system object/prototype must contain within it several **collections of objects** of different types:
--a generic particle class ('generic' means one single class supplies all parameters needed for any/all kinds of particles without modification). Every particle object has its own position, velocity, mass, force accumulator, and more, and is compatible with any kind of force, constraint, and behavior rules without re-naming or revising.
--a generic force-maker object type (whose objects can describe a gravity, a spring, a tornado, a charge, etc); and
--a generic constraint object type (whose objects can describe wall, a rope, a rod, enclosure, distance limits, etc).
Member functions must include a generic 'force-applying' function, a generic 'dotMaker()' function, a generic solver() function, a generic constraint-applying function, and a generic render function.
PLEASE FOLLOW the 'Weekly Goals' guide below to organize your program!

f) – Create 4 different particle-systems objects (e.g. 4 separate variables of type PartSys). Set their parameters to create and show at least **four different *kinds* of animated particle systems shown and run on-screen at the same time.** Each system has its own separate rules of movement, with user-choices for different behaviors and integrators. These 4 different *kinds* of particle systems are not required to interact, but I encourage it – interactions permit more interesting simulations (e.g. waving flags that disrupt or modify blowing smoke), and your use of 'generic' components (state variables; solvers; particles; force-makers; constraints) simplifies the task.
At a minimum, your project should include include:
Particle System 1)--At least one system of at least 500 particles that move independently (e.g. blowing leaves, fireworks, leaves of grass, herds of people) under influence of **a shared, position-dependent 3D vector field** of forces defined in the 'world' coordinate system, in addition to ordinary gravity. The vector field must vary with position, may be the sum of several separate, simpler fields, and it can be either fixed or changing with time (e.g. a tornado-like wind

field that wanders around; a 'blower' force field that pushes forces away from the mouse cursor on left-click, or attracts them on right-click). HINT: User-controlled, interactive fields are far more fun—steer the tornado, make a user-aimed fire-hose that squirts particles, etc. HINT: electrostatic forces that (under user control) attract or repel particles to a movable 3D object can lead to very interesting behaviors when added to wind-like forces.

Particle System 2)--At least one system of at least 90 particles that act together in loosely associated ways to **exhibit 'flocking' or 'boids' behaviors** (must exhibit all 4 'boids' behaviors: separation, cohesion, alignment, and evasion). Note that each particle influences all others nearby, and that you can 'steer' a flock of 'boids' interactively by adding some gentle forces to all members of the flock. HINT: to demonstrate the 'evasion' behavior, you may wish to add some simple building-like constraints (e.g. cylinder, rectangular solid, sphere, torus, etc.) to act as obstacles along the flocks' flying path; often flocks will split into separate streams that flow around and through the obstacle without colliding with it. Note that our textbook's Chapter 6.5 gives a detailed, step-by-step description of flocking algorithms.

Particle system 3)--At least one system of at least 600 particles that simulate continuously-burning flame and fire-like behavior on-screen, following the methods described in the Reeves et al. 1983 paper (see Canvas), in which each flame particle has a visible life-cycle that affects mass, color, trajectory and possibly other effects.

Particle System 4)--At least one system particles **linked together by a network of springs to form one or more stiff yet elastic 3D semi-rigid shapes of at least 10 particles each, that users can manipulate interactively**. Your interactive shape could be a 1D-connected stretchy, rope-like chain, a 2D-connected rubber-cloth-like surface, a 3D-connected deformable body, or combinations. Users must be able stretch, squeeze, drape bounce, and/or otherwise manipulate your 3D spring-mass shape using fixed constraints, moving constraints, user-controlled constraints or any combination. (NOTE: all your particle systems, even 'ropes' must be computed in 3D. I will not accept 2D-only results such as this nice (but too-limited) 2D-only tear-able cloth: **http://codepen.io/suffick/pen/KrAwx** (Note its nicely usable editor… try it!), or this 2D 'sticky thing': **http://www.spielzeugz.de/html5/sticky-thing/**

Show us something interesting – the possibilities are nearly endless! For example, from a 1-D spring-mass network you could construct a long elastic 'rope' or 'ribbon', each end attached to a different point above ground-level on a different wall, that settles into a catenary curve under the influence of gravity but permits the user's mouse to 'drag' any point on the rope to stretch to any on-screen location, then 'release' the stretched rope to snap back violently, eventually calming itself to return to catenary shape. A row of stiff 'hairs' attached to a handle could form a flexible paintbrush A 2-D spring-mass network can form a flexible sheet that emulates a flag that hangs from a fixed (or movable?) flagpole, and users could 'fly' the flag with controllable 'wind' (a turbulent 3D force-field). They could also disturb it with 'puffs of air' triggered by nearby mouse-clicks, or choose to release the cloth to fall down by gravity and to drape over a sphere or cylinder. Eventually, the 'flag' may slide off to form a heap on the floor. Or perhaps you could make a 'trampolene' (a spring-mass sheet fixed at all edges) and drop a sphere (a moving constraint with mass) to bounce on that surface. ?Could you make a tennis racket?
For a 3-D spring-mass network, you could construct a tetrahedron or cube using springs as edges (you may need a diagonal internal spring to prevent cube collapse) or 3D star-like shape that users could toss upwards to fall and bounce raggedly off walls, floors, and other obstacles. Think broadly; devise interesting examples of your own!

g) – Your particle system class must offer at least **two separate integrators or 'solvers' for each particle system** type, and enable users to switch between them as they run, without forcing users to stop or re-start the simulation (as in the starter code):

-a fixed-timestep **explicit integrator**, such as an Euler, Midpoint, or Runge-Kutta method (which might be unstable and oscillate/'blow up' in stiff, strongly constrained systems), and
-a fixed-timestep **implicit or semi-implicit integrator** such as Verlet, semi-implicit Euler, or various fully-implicit back-wind methods (Euler, Midpoint, etc) which will be stable, but may appear over-damped for large timesteps. You must demonstrate your project using FIXED timesteps, but you are welcome to experiment adaptive timesteps as an added/extra credit feature: warning--they're neither necessary nor advisable.

OPTIONALS: Your program **MAY** have:

1) Any additional particle shape, rendering methods, movement, motivation and behavior rules you wish. You have my permission (and encouragement) to break the laws of physics, to do the physically impossible, and to amaze us all!
2) As many particles as you wish, rendered in any shape or material you wish (e.g. a dot, a disk, a texture-mapped maple leaf, a fading streak of fireworks, etc.), but demos MUST stay interactive: >3 frames-per-second(FPS)).

3) Collision detection between particle-driven objects.
This can help make cloth 'drape' properly, make rubber balls bounce around in interesting ways, etc.
4) Any lighting and rendering method you wish.
5) Adaptive time-steps, but I don't recommend it. Are you sure you need them? Implicit or semi-implicit solvers are much simpler, less quirky to debug, and always work well, unless their intrinsic 'damping' disrupts your system's goals, and your system is very, very 'stiff'. All 'compulsories' above require fixed time-steps.
6) Your program DOES NOT have to run at high frame rates – 3FPS is enough; show us during demo day!
Time spent tinkering with code for better speed is time you could be spending on making more interesting movement, shape, rendering and interaction. I would much rather see a more sophisticated system with a few tens of particles that run and react slowly than a fast system without any interesting content. If you wish, you may make movies of your particle system if you feel it is too slow to show in real-time. Later you can use your ray-tracer (Proj B) to make the movie as beautiful as possible.

Weekly Goals:

(Remember, these are minimum weekly goals, assignments imposed to help you accumulating a tragedy at the end of the quarter. To do well in the course, and for the most pleasant and productive progress, stay well ahead of these minima. Give yourself time to explore, invent, and investigate any quirks you find. All of the most visually interesting results are later, piled up and spectacular as you approach Week 4 Goals.
Reaching the Week 4 goals before week 4 ensures you have more time for the 'good stuff'!)

Week 1: **'3D Basic Ball Bounce' code.** Use CS351-1 'starter code' and explicit (Euler) integration to 'throw' several balls into a 3D room. Let users position their camera in 3D above a vast ground-plane grid, let them choose where to start the ball, where to aim it, and how hard to 'throw' it. Make that bounce off walls of a 3D box, follow parabolic paths due to gravity, and bounce off any wall, floor, or ceiling they may hit. Try out larger/smaller time steps, lossy bouncing (reduce velocity slightly with each bounce), and air-drag (a force opposite to velocity). Expand it to 'throw' a steady stream of hundreds of balls: store all the parameters of those balls in one giant interleaved JavaScript array.

Week 2: Expand to 1,000 bouncing balls into state-variable form.
a)--Devise state-space particles that include its 'force accumulator' vector where find the sum of all forces pushing on the particle in its current state, and a collection of particles in single 'state variables' s0, s1, s0dot; each one a Float32Array in JavaScript. Why have a 'CPartSys' object/prototype? because it encapsulates the entire particle system within one complete unit without any other external storage or functions to operate correctly. With this encapsulation, you can (soon) write programs that contain multiple independent particle systems, by simply declaring multiple 'new' instances of CPartSys objects, as required by Project A.

b)-- Next, devise a CForcer: an arbitrary/generic force-applying object (later we will make an array of them).
Caution! do not make forces into members of your particle class, and do not make them part of your state-variables. You'll soon realize why not:

- because a force may need to apply to only a sub-set of particles (e.g. springs, flocking forces), and that sub-set may change over time;
- because we may wish to create/destroy and enable/disable forces interactively.
- because we must be able to make sensible time-derivatives of all elements of our state-vectors (e.g. s0), but we don't need (and often cannot easily compute) the time-derivatives of these forces.

Instead, make a CForcer object for each force-causing entity in a particle system. Each CForcer object specifies how to compute the forces it creates, and specifies the set of particles affected by those forces. For example, a 'gravity' CForcer

object would probably apply its forces to all particles, a CForcer object for a single spring that connects 2 particles would apply equal and opposite forces to those two particles alone, and a force that causes 'flocking' behavior of 'boids' such as separation, alignment, cohesion, and evasion (see Chapter 6.5 reading) would apply forces to each particle in the 'flock' in response to a continually-changing set of nearby particles, chosen by their distance.

c)--Then create an collection of CForcer objects that I will call 'CMotive'. I like to use an array of CForcer objects, but you might prefer another way. Whatever you choose, be sure to make your 'CMotive' collection of CForcer objects a member of the CPartSys class or equivalent.

d)--Create the vitally important function applyForces(), a member function of CPartSys class or equivalent, accepts just 2 arguments: the CMotive collection of CForcer objects, and a state variable 'S' (careful! Don't assume s0 is the only possible argument here!). One-by-one, for every particle in S, this applyForces() function finds the total of all force vectors applied by all the CForcer objects in CMotive. In other words, the forceMaker() computes the 'force-accumulator' values for every particle in state S, using every applicable CForcer object held in CMotive.

d)--Create the 'dotMaker()' member function, part of the CPartSys class or equivalent, that accepts *ANY* state vector **s** (complete with the sum-of-forces computed by applyForces()) and computes its time-derivative state sdot.

The dotMaker() function applies Newtonian physics. It converts the sum-of-all-forces on each particle into its acceleration (f = ma, so a = f/m). Acceleration is the time-derivative of velocity; in the vector position where s0 stored velocity, we make the s0dot vector store acceleration. Similarly, velocity is the time-derivative of position; in the vector position where s0 stored position, we make the s0dot vector store velocity, which we simply copy from velocity stored in s0 vector (See class notes for details).

e)--Create the 'solver()' member function, part of the CPartSys class or equivalent, that creates a new state s1 from current state s0 and its time derivative s0dot. For a simple explicit or 'forward' Euler-method solver, compute s1 = s0 + h*s0dot, where 'h' is the timestep. HINT: instead of MKS units (meters, kilograms, seconds), begin by using 'frames' to measure timesteps, where h = $1/30^{th}$ second = 1 frame. If you measure time in units of frames instead of seconds in your first, early Euler-method solver, then you can then eliminate the time-step multiply.

f)--Create a render() member function within the CPartSys class to render the particle system described by state vector s0. For now, just use WebGL's 'point' primitive (and try the Fragment Shader trick in Week1 starter code that makes a point look like a 3D sphere), but later you can expand this function to render particles as teapots or other shapes, or as little texture-maps with changing images on them (great for smoke!). Here you can add lifetimes to particles (so particles are created and destroyed, and change over time), and draw particles that leave 'trails' behind them (e.g. for grass). Try to make a Reeves-like 'flame' system…
HINT: at the end of a particle's lifetime, don't change the amount of memory used (SLOW!), but instead just change the state of a Boolean 'isAlive' member variable. Other functions can then re-use and re-emit this particle object from a new location along a new trajectory.

g)--Create an animation within CPartSys class. After main() initializes all the particles inside the 'current state' vector s0, your draw() function should ask each particle system to:

1. call applyForces(mot, s0) to find the sum-of-forces for each particle in s0, applying every force from your collection of forcers.
2. call dotMaker(s0,s0dot) to compute time derivatives s0dot from current state s0,
3. call Render() to draw current state s0 on-screen.
4. call solver(s0,s0dot, s1) to find new state s1 from current state s0 and its time derive s0dot.
5. swap new and current state: s0 becomes s1, s1 becomes s0 (and s1 is old and longer valid).

Week 3: Your first **coupled particle system**. Expand your CForce class to make damped springs, and use it to make a 'snake' (a 1-D line or ring of particles connected in a chain). What happens when you make springs then a 2D sheet of

particles, etc.? After the first system works, re-examine your classes; can you make them simpler and more elegant? EXPECT instability if your springs are 'stiff' and your time-steps are large; reduce the timestep and reduce the spring forces and/or add velocity damping to reduce instability. Consider writing a system where you compute multiple time-steps between on-screen display.

Week 4: Advanced Solvers, Multiple Systems Revise, expand your solver; expand it to allow user-selection between EXPLICIT (Euler) and IMPLICIT or SEMI-IMPLICIT solvers for the same system, changing nothing else. Note hugely improved stability-- you'll need that stability for coupled 'stiff' systems! Then you can make systems with stronger springs, less 'damping' for those springs for stiffer objects, and even add some 'hard' constraints (from reading) that won't cause instability.

Week 5: Multiple instances of particle systems. Get more than one particle system on-screen at once.
Finishing touches: improved rendering (e.g. texture mapping, better lighting, streaks, etc.)

Sources & Plagiarism Rules:
Simple: *never* submit the work of others as your own. You are welcome to begin with the book's example code and the 'starter code' I supply; you can keep or modify any of it as you wish without citing its source. I strongly encourage you to always start with a basic graphics program (hence 'starter code') that already works correctly, and incrementally improve it; test, correct, and save a new version at each step. Also, please learn from websites, tutorials and friends anywhere (e.g. .gitHub, openGL.org, etc), please share what you find on Canvas Discussion board (but NEVER post code you will turn in for grading—only ideas or examples), but you must always properly credit the works of others in your graded work—**no plagiarism!**

Plagiarism rules for writing essays apply equally well to writing software. You would never cut-and-paste paragraphs or whole sentences written by others and submit it as your own writing: and the same is true for whole functions, blocks and statements. Never try to disguise it by rearrangement and renaming (TurnItIn won't be fooled). Instead, study good code to grasp its best ideas, learn them, and make your own version in your own style. Take the ideas alone, not the code: make sure your comments properly name your sources. (And, Ugh, if I find plagiarism evidence, the University requires me to report it to the Dean of Students for investigation).