Real-Time Visualisation Summary

"Usually on the exam"

Real-Time Molecular Visualization

Visualization on a molecular level, i.e. cross section of a bacteria
because i.e. DNA, molecules cannot be seen with microscopes
How to visualize?
painting ? lots of work (lots of repainting)
computer support - description with rules and generated
size: (protons, .. not interesting) < nano (atoms visualized of spheres) < meso < micro (cells)
Small molecules - solvent (water, etc.) - ligand (messenger) - lipids (form membranes)
Larger molecules - Nucleic Acids (dna, rna), sequence of molecules (i.e. 4 bases form a message); protein chains of amino acids (dna);
How do molecules look like? mesoscale - microscope; atomic structure - x-rays => combine for atomistic model
How to create them? Surface container → (molecular ingredients) pack them overlapping)
Optimisations:
Procedural impostors - billboards: render a quad - calculate depth for each pixel → correct z order; emitting geometry (instance on the gpu, only use position);
Level of detail - less detail the further away (use i.e. k-means to cluster molecules → represent them as sphere; or affinity propagation, hierarchical)
Approach for Nucleic Acids - defined through cubic spline (reparametrize it; place points in regular intervals) - calculate tangents (use principal direction) - rotate tangents (same angle as bases and dna) - place molecule along the curve (tangents) → use geometry shader; only parametric curve representation necessary
Occlusion Culling - i.e. use hierarchical z-buffer; mip-map levels are created; compare a molecule with lowest resolution of the z-buffer
Illumination screen space AO or object space AO; (2 level space screen space AO is good) to create shading between two compartments (shows depth better)
Multi-Scale models use colors for departments for rough scale (adaptive coloring)
Multiscale ID Buffer render buffer in a fragment to the ID buffer - i.e. type of molecule, do it for several levels; look up the buffer and determine by the distance which color is used
use color wheel (the more parts of one type are shown the bigger the color range is)
Multi-Instance Cutaway Visualization choose which molecules are in the scene (increase their amount with the slider i.e) multiple cutting planes, spheres, rectangles, view aligned cutaway,
Multi-scale and Multi-instance Labeling use again an id buffer (label levels buffer - representative label (choose rules how to pick representatives i.e. well visible, centered)

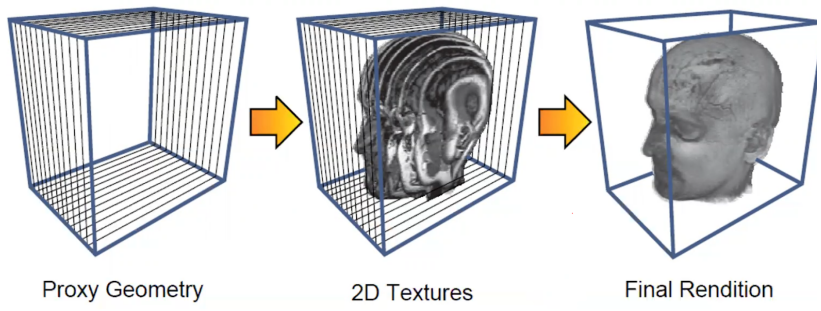
General Purpose Computing on GPUs

Perfect fit for everything that can be parallelized (i.e. machine learning)
Graphics Processing Units, flexible massive number of FLOPs, not as easy programmable

Host = CPU, (Compute) Device = GPU; Host program on CPU sets up GPU (manages memory, schedule); Kernel program (run parallel) on GPU;
Global size (i.e. full image size) > Work Group (one Block) > Work Item/Thread (instance of kernel)
Host Memory → Global / constant memory (GPU) → Local (shared) memory (for each block) → Private Memory (each kernel / work item)
Single Instruction, Multiple Thread Commands are executed concurrently (collection of 32 threads); branching causes thread disabling etc.
CUDA kernel function needs entry point from host __global__ void kernel(params…)
call with kernel<<<grid,block>>>(params); identify “which items are yours” by getting threadId, blockId etc.
GPU Profiling is a bottleneck search; performance depends on memory transfer, memory throughput and instruction throughput
Sequential access
Cache line is loaded (128 byte); each k-th thread accesses every k-th 4 bytes;
problematic if each warp accesses the next line instead of its own (twice the amount of data loaded);
worse: strided access (wants to access every x-th entry) → multiple lines need to be loaded;
Shared Memory
All of the memory can be accessed (caching does not matter so much); Problems if threads access the same part of the memory (sequentialization)
if all threads want to access a single data it is broadcasted and not a problem
Warp divergence affects instruction throughput (some threads will do different “things” and others go a different path (i.e. a if condition) → different paths within the same warp should be avoided
Streams, Concurrent Execution Several operations can operate concurrently; split data into parts and execute (faster than long loading and executing once)
Optimizing Parallel Reduction
Addition? Divide and conquer. Problem? Global Synchronisation. Tree based approach used within each thread block;
Use Kernel Decomposition: Call kernel multiple times (level 0: 8 blocks), (level 1: 1 block)
Goal Reach GPU peak performance, choose right metric (Gflops, Bandwidth); reductions have very low arithmetic intensity;
Benchmarking is important (right metric) → to improve the solution (right functions? data accessed efficiently? etc.)
Reduction - Interleaved Addressing each thread loads one element from global to shared mem → sync → do reduction in shared mem → sync → write result for this block to global mem
but there is a problem: highly divergent (with if statements), % operator is slow
Solve this by strided access → new problem (shared memory bank conflict)
Sequential Addressing → thread based indexing; Problem: Half of threads are inactive → halve the number of blocks and each thread loads 2 data instances → unroll last warp → completely unrolled (outdated apparently, but takeaway: try out what improves performance)
Atomic Operations read-modify-write atomic operation on one 32-bit or 64-bit word residing in global / shared memory; guaranteed to be performed sequential

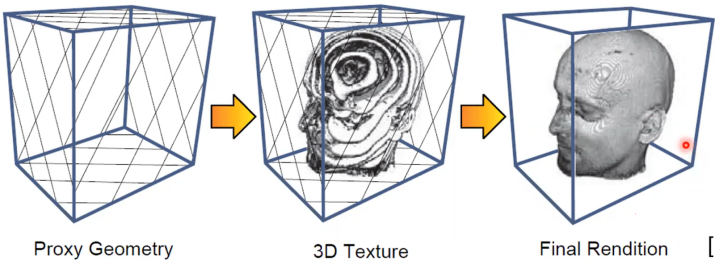
Real-Time Volume Rendering and Illumination

Simulation of light propagation in volumes; problem? interaction light and media (volume)
3D scalar fields obtained from - measurements (CT), Polygons (rasterized to voxels), Procedural models (set of rules to create volume), simulations
Volume Rendering Techniques 2D visualisation (one slice of the volume), Indirect 3D visualisation (isosurfaces - show surfaces that represent a region with densities within an interval - marching cubes to create), direct 3d visualization (DVR -- volume emits light → use physical properties to display data)
Isosurfaces good for numerous & complex objects and labeled objects
Volume Rendering good if segmentation does not work, low data contrast, thin objects
Real-Time Volume rendering different from surfaces (meshes), larger number of data, interior of objects; segmentation masks (tagged volumes), no empty spaces - light interaction with all voxels; use Ray Casting (image order) vs. Voxel projection (object order), slicing splatting; effects like Global illum, DoF make effects slow
Lighting propagates in a line until it hits something; Absorption, Emission and Scattering
Sample to solve integral equations (hit-point to exit-point → summed up = pixel on the screen) → sampling rate defines quality (higher = better);
Compositing
Sample points not always in the center of voxels → interpolate voxels
Contribution for each point (all colors summed * all transparencies multiplied)
From back to front vs. back to front; Max vs. Min vs. Additive (density) etc.
Transfer functions map data values to color and opacity; Classification (make histogram → determine from the data the different materials → estimate for each voxel percentage of material → each interval represents a material having a color etc.
Volume Rendering Pipeline
Data Traversal → choose sampling points (discretization); Interpolation →3D grid; Gradient Computation; Classification (map data properties to color opacity); Shading / Ilume (local global); Compositing (iterative computation)
Ray Casting
Natural image-order method; evaluate equation along the ray (no secondary effects i.e. shadows); Cast a ray through the volume → Each point interpolated (of neighbouring voxels) → Compositing (sum together values of points); optimise with empty space skipping (quad trees), early termination if sufficient opacity; subdivision; adaptive sampling; ray packets traced together
Slicing
2D texture based (axis aligned), 3D texture-based (view aligned); object order approach; uses a proxy geometry for slices



Slices need to be according the view direction (not parallel); Aliasing artifacts (if sampling rate is too low); three stacks needed

View Aligned Slicing



less artifacts, better image quality; easy increase of sampling rate to create more slices;
Slicing Advantages Utilize conventional GPU pipeline, Global illumination with forward scattering can be integrated; depth-of-field effect can easily be integrated
Disadvantages no empty space skipping, no early termination, fixed compositing direction
Splatting object-order approach; similar to slicing; show voxels as spheres (certain radius) → represent physical properties of the volumetric data
Axis-aligned (artifacts) vs. view aligned spalatting (better)
Illumination
emission-absorption model (no reflection, scattering, etc.) vs. scattering (BRDF, more realistic, but slower) - for volumes phase functions are used
Single scattering vs. Multiple scattering (global illumination)
Optical properties of matter opacity is important (rep. light absorption) can be represented as color → transport color (defines how material changes the color) i.e. wax block light from top darker to the bottom
Phase Functions descriptive optical property; describes scattering for whole sphere (not half as brdf); direction of the phase function → color changes depending on the angle its scattered
Bounces of Illumination = light incidence reflection; only one bounce is unrealistic (no ambient light, no color bleeding, no scattering, etc.) → multiple bounces hard to compute; second bounce already contributes to the shading of each sample along the ray; optimisation (enclose sample with sphere and compute contribution only within sphere = early termination), summarize inside spatial regions instead of multiple samples
Half-Angle Slicing compute illumination same time as rendering happens; two buffers - rendering + illumination; slices are necessary; only one light source possible
Idea: Define two vectors (viewer dir, light dir define the slicing direction (half angle) = slices are orthogonal to this vector); slice closer to the eye inherits illumination from previous slice; similar to view direction (both need to be corrected)

Compute current slice illum by sampling previous illum buffer; shade samples with illumination; composite the shader result to the rendered image (back to front)
Changing view direction → recomputation of everything (flip v if necessary to get a sharp angle between v and l);
Algorithm creates sharp shadows, only one traversal, four additional buffers (2 copies of 2 buffers); buffers are swapped each slice (sync on CPU level)
Soft shadows can be realized with gaussian blur kernel (multiple samples of previous slice)

High-Performance Visualization

many equations, large dataset; Client interacts with a cluster (parallel computing);
Inter component communication necessary (send data, geometry, images (rasterization) to achieve functional- and data- parallelism
Data characteristics depending on the data (data size and type) choose correct representation and methods
Pipeline partition Data - Visualization - Render - Display; complicated data, complex computation → only display on client; small datasize, high interactivity? → only data on server; all other cases? data and visualization on server;
Send Image upper bound on data (image always the same size); con: interactivity needs resending; minimum amount of network traffic
Send Data and Send Geometry Rendering can be done using WebGL on client; size of data or geometry is relatively small; interactivity is important;
Load Balancing distribute tasks evenly to nodes; Static? size of work predetermined, low communication overhead → load imbalancing; Dynamic? Size of work determined during execution → better balancing but high communication overhead
Parallel Rendering Taxonomy Parallel rendering as a sorting problem (by distance from camera); Sort First (sort primitives i.e. spheres, then send to processor), Sort Middle (sort screen primitive after geometry processing; sort tessalized triangle); Sort Last (redistribute pixels)
Sort First Pre-Transform (which object will be in which bucket) → Bucketization (one bucket for each processor) → Redistribution(right order for objects over multiple buckets) → Geometry → Pixel Rendering
Pro: Low communication if tessellation ratio is high, oversampling ratio is high and frame-2-frame coherence can be exploited; processor implements entire rendering pipeline for portion of the screen;
Con: Initial distribution on processors, primitives may require different amounts of work (size dependent); Certain areas will have more primitives (to counteract make more regions, multi-region responsibility); to take adv of frame-2-frame coherence data handling code may be very complex
Communication cost = const factor * nr. of primitives * overlap factor * average size of prim.
Good when high coherence
Sort Middle similar to sort first; Geometry → Bucketization → Redistribution → Pixel Rendering;
Fix clumping by make more regions → leads to more overlap
High communication cost if tessellation ratio is high; Geometry is sent every iteration
General and straightforward; better than sort first if tessellation is low and low frame coherence;
Communication cost = nr of screen primitives * overlap factor * average size of screen prim.

Good when Low coherence, low tessellation
Sort Last Sparse only sends pixels that were created; depends on nr. pixels generated
Communication cost = nr primitives * average prim size * samples pixel
better than FFM unless depth complexity of entire image greater than nr of processors or num processors * resolution < num primitives * average primitive size
Full Frame merging full image is sent; dependent on nr. of processors and resolution
→ Uneven distribution of rendering work, suffers less from primitive clumping, sparse merge network traffic can be unbalanced;
Pro: renderers implement full rendering pipeline and are independent until pixel merge
Con: Pixel traffic might be high, especially when oversampling, transparency
Communication cost = num processor * resolution * samples pixel
Good when complex primitives covering small area
Direct Send each processor responsible for one part of image; after calculating the pixel it sends to the correct processor, cost = p * (p - 1); hard on network (nodes might not be connected directly)
Tree Send basically divide and conquer, load imbalance (i.e. PE1 on top of pyramid very busy, others idle)
Binary Swap instead of tree send, P1 sends primitive to P2, P2 sends primitive to P1 (left and right half), step 2 = p1 and p3 exchange top part, p2 and p4 bottom part; final → each P has info about its own region;
Log(p) stages necessary; pro: good load balance, con: 2^n processors needed
2-3 Swap each number of processor can be written as a sum of 2s and 3s;
Build Tree to get log(d) depth for the tree, that's why you check vs 2^d, start with $2^d < n < 2^{d+1}$ recursive calculation, tree needs to stay balanced
Radix-k only do compositing within the group (first horizontal groups, then vertical groups)
better than binary swap
In Situ Processing simulations can generate large quantities → in situ should show intermediate results
Co-processing tightly coupled, synchronous, vis routines have direct access to simulation code memory, memory constraints, large impact on simulation (crash / corruption of memory affects simulation)
Concurrent Processing loosely coupled, vis runs on dedicated concurrent resources (crash does not impact simulation), access data via network (expensive IO), data movement costs, requires separate resources;
Hybrid data is reduced via coprocessing and sent to concurrent resource; complex and shares negatives of other approaches
Z-cure data organisation
Find linear structure that connects all neighbouring cells in the storage (preserve locality)
(similar to quad tree, lets you efficiently find neighbours)
use bit counting (position is a binary number)
Pro: not based on row or column, better I/O performance when subset is read
Con: connection of last cell (of first field) to first cell of next field
Hilbert Curve (it's a lot better than Z-Curve, trust me)
Exascale → algorithms need to be changed, power issues (can't just increase compute power)
blue waters i.e. would need 1.5 gigawatts to cope with exascale data
Physical Fusion, Climate Science, Traffic Simulation
also useable in InfoVis

Real-Visual Analytics and Web-Based Visualization

automated analysis techniques with interactive visualizations for very large / complex data
Volume of data, Variety, Velocity (streaming data) and Veracity (Uncertainty)
Big Data tall data (billions of records) and wide data (thousands of variables)
Perceptual Scalability display res < nr of data points, perceptual limitation (can't see everything at once)
Interactive Scalability inefficient queries, data processing, rendering performance
Scalability affected by human perception, resolution, visual metaphors (can't see each different color etc.), interactivity, data structures, computational infrastructure (network etc.)
Overplotting occlusion, clutter, readability limited and resources wasted (because they cannot be seen)
Alpha blending: reduce opacity of data points - dense regions visible, outliers vanish
Filtering: Selection of data set (i.e. tuesday), con: user needs filter criteria that is suitable
Sampling: Random selection of data subset, outliers may be omitted
Summary Statistics (model based abstraction) instead of individual data items compute and show statistics (i.e. confidence intervals)
Be careful what abstraction is used (i.e. box plots don't work if no normal distribution is present)
Density / Aggregate / Binned Vis data binning, transfer function (value → color) etc;
i.e. Histogram, Bar, Line, Choropleth map for 1D, binned scatter plot, heatmap, temporal for 2D; can cause aliasing effects (i.e. nr of bins affect quality)
Kernel Density estimation for each point x calculate sum of deviations of samples
width of kernel changes probability density function; different kernels possible (gauss, rect)
Violin plots (box plot + density estimation)
Information Seeking Mantra Overview first, zoom and filter, details on demand
Hierarchical Aggregation aggregation of data based on hierarchy (i.e. choropleth maps for elections), adjust bandwidth for density scatterplots
Sampling local overplotting, missing outliers vs. **Heatmap** global patterns, local outliers
Takeaway points Aggregate vis very scalable, interactivity challenge (100ms interactive response limit for brushing and linking (filter) panning / zooming (resolution change)
Latency sources query, data processing, rendering
Interactive Scalability random sampling? ok but not so good
Porgressive analytics partial results of computations under user bounds that converge to final result, user can interactively change parameters)
Confidence Intervals sample standard deviation depends on mean → full recomputation necessary; but there are some online methods
Multi-threading on Client no UI blocking etc.
Parallel Data Aggregation thread takes a section of input, etc. CUDA
Precomputation whatever is possible!
Aggregate Queries data cubes: ndarray, quick summaries of data, information split into independent variables and dependent variables, aggregations are projections i.e. x-axis = product, y-axis = country, z = amount
Size is problematic for data cubes (very large)
Reduce dimensions of data cube, i.e. instead of 5-d data cube use 4 or 5 3-d data cubes
Data Tiles i.e. google maps, decompose canvas into static tiles; loaded dynamical; can be used for any vis that is zoomable

Graph Tiles Compute hierarchical aggregation using communities; community = strong bonds within (density) and less density between communities (meta nodes)
layout the highest level then lower levels layouted within the communities, etc.
can be parallelized
Positions and links (+ annotations) can be stored (precomputed) and only rendered on the computer (allows filtering etc.)
Prefetching while user is interpreting, use the time to preload next data tiles
Efficient Rendering d3 easy event handling, visual scalability but poor performance
2D canvas is complicated and also not very well performing, webgl is so-so complicated but very nice performance → GPU goes brrrrrrrr
raycasting for selection (use quadtree to find closest point faster)
Image Space Visualization Operations transform data tuples to RGBA pixels and add color mapping, glyph rendering etc.
Data packed as RGBA images; query fragment shader to sum bins and write into FBO, render fragment shader determines bin and color for each pixel
Summary Scalability
Perceptual problems: Overplotting, visual clutter; Solution: Render models orr aggregations
Interactive problems: too much data queried; Solution: progressive queries, aggregate queries/vis;
Too many graphical markers, Solution: Parallel aggregation, data tiles with prefetching

Real-Time Fluid Simulation and Visualization

Challenges Modeling continuous fluids need many approximations, topological variations, various interacting objects, numerical stability/accuracy;
Performance: grid and temporal resolution
Artistic control: parameter tuning? model many types of fluids?
Simplification: no change in volume of fluids (incompressible)
Navier-Stokes Equations analytically only simple cases can be solved, in CG just approximations;
Change in velocity = advection - pressure + diffusion + external forces
(non-linear partial differential equations)
Incompressibility constraint = triangle pointing downwards * u = 0
Advection velocity causes transport of objects, densities, etc.
Pressure incompressible fluids, pressure causes distribution of acceleration
Diffusion objects, densities get distributed to neighbour cells (models the viscosity, honey = high viscosity; does not dissolve easily in water)
External forces gravity, pistons, etc.
Lagrangian Approach
Liquid is represented as particle (one particle has 1000s water molecules); compute forces between them; Models the forces according to certain predefined smoothing kernels
Pros: Mass/Momentum conservation, intuitive, Fast, no linear system solving
Cons: Suffers difficulty with pressure and incompressibility, connectivity information/surface reconstruction, time step needs to be small to ensure stability
Eulerian Approach
Liquid is divided into a grid; keep a status for each of these cells;
Define scalar & vector fields on the grid; operator splitting technique to solve each term separately

Pros: Derivative approximation, adaptive time step, ensuring incompressibility is easier
Cons: Memory (whole liquid is stored as grid) usage & speed, grid artifact / resolution limitation

Smoothed Particle Hydrodynamics (Lagrangian)
Pressure computation → Kernel Functions - models the smoothing (define weight of contribution from neighbours) → boundary condition - what happens on the boundary of the box of liquids → nearest neighbour search (spatial data structures) - surface reconstruction (3D scalar fields, marching cubes)
Advantage: Model solids also as particles
Eulerian Approach - Domain Discretization
Space needs to be divided into cells (derivatives must also be derivatives)
Center of difference scheme is used to approximate derivatives (look one to left, one to right and then divide it)
Cell attributes are stored in 3D textures; in each step update values by running computational kernels over the grid domain;
Helmholtz-Hodge Decomposition a vector field can be decomposed into u + derivative p; u can be used for incompressibility constraint (because it has 0 divergence)
3 computations (advection, diffusion & force application) results in no zero divergence
??????
Poisson equation gives non divergent velocity field (maybe)
??????
Pressure Projection use again the H-H decomposition; projects non-divergent field on the
??
lecturer is also lost in the equations
??
probably this won't be on the exam
??
Simulation Loop don't solve equation at once, but compute one after another → addition is substituted with "operators"; each step is applied on a vector field;
Advection solved by forward euler; considers cell as a particle (direction where the particle would move) and move the particle along the velocity → is actually bad because it can "blow up" -- accumulating errors;
Better: Semi-Lagrangian: look where the particle got here; look in the negated velocity direction;
Even better: MacCormack Advection fixes smoothing and loss of detail from semi-lagrangian
Diffusion models the thickness; measure how resistive a fluid is to flow; very """simple"""
External Forces local or body forces, gravity, fan blowing air, …
Pressure Projection (again) solve poisson equation for pressure scalar field, subtract the scalar field from the divergent field to get a non divergent field → fluid incompressible
i.e. Jacobi iteration, conjugate gradient, multigrid methods
Boundary Condition divide into interior and boundary cells; no-slip velocity boundary condition (velocity on boundary is zero) or derivative on boundary has a certain value; one pixel border for storing boundary values;
Dynamic obstacles are related to this;
Voxelize the objects using a "D array of stencil buffers
Staggered Grid pressure and temperature are in the cell centers, velocities on the edges → prevents artifacts

Visualization quantities are visualized (temperature, density, etc.) use volume rendering techniques (i.e. raymarching);
Smoke store temperature (a gaussian splat injected to a color dye texture
Fire similar to smoke; reaction coordinate (keeps track of time since ignition, value less than zero = fuel exhausted), map reaction coordinate values to color
Liquids Level Set Method, each grid cell gets a shortest signed distance??
Conclusion
Fluids modeled by Navier stokes equations - advection, diffusion, pressure and external force; Two major viewpoints = eulerian and lagrangian
Advection methods = forward euler, semi-lagrangian or MacCormack; Pressure projection
Boundary condition and dynamic obstacles; Visualization and rendering of fluids

Graph Visualization

Graph = Nodes (items), Edges (Relationships) i.e. weighted or unweighted
Clique (edge between all nodes), Connected components, …
Geodesic distance (shortest path between 2 vertices (how many edges are needed)
Types of Graphs directed, undirected, mixed, tree, directed acyclic graph, networks
Hypergraphs an edge can have multiple nodes (not only 2)
Graph Preprocessing filter graphs (remove irrelevant parts), aggregation (i.e. clique as one node)
Graph Representations Node-Link-Diagram, Adjacency Matrix, Combination, Treemap (windirstat), Information cube (nested 3D representation)
Node Link Diagramme
Layouting - position them in space (Fore-directed, Circular, Arc Diagram, Layered)
Edge Bundling - similar edges will be combined to bigger edges (can be simulated/calculated with electrostatic forces → virtual vertices created along the edge, repulse to other nodes on the same edge and attraction towards vertices)
Focus+Context - interactive, simplify nodes/edges not in context (i.e. magnifying glass, or abstraction (gray out)
Deterministic Layouts
Circular Layouts - nodes in circles, edges through the middle, filter, etc.
Arc Diagram nodes in a line, circular edges
Non-deterministic Layouts Force-directed layout, basically a simulation, repulsion (for unconnected nodes i.e Coulomb's Law force decreases with distance)) and attraction (between connected nodes i.e. Hooke's Law spring force)) and Focal point (i.e. attraction to center, grows with distance; or several focal points for clustering)
Euler Integration to compute layout (per frame), Acceleration (force / mass), new velocity = velocity + acceleration * timestep), new pos = pos * new velocity * timestep
Error is proportional to the timestep;
Verlet Integration - express velocity as current - previous position → slower than euler but much more stable);
Complexity attractive forces = O(n) for each edge a force, repulsive = $O(n^2)$ every pair of vertices repulses each other
N-Body Simulation optimizations
with Quadtree - space divided into tree structure; if two vertices in the same cell → split cell into 4 cells (represented in the tree as well)

Quadtree can be used for nearest neighbour simulation; as largest force is applied to close neighbours, and little force to far away nodes;
Task: Find all vertices within the circle; Check if root intersect with radius, then check all children; if leaf check distance with node it self;
→ takes O(log n) time
Barnes-Hut what if far away (many forces) vs. one close (strong force); forces would not be found with nearest neighbour;
Steps: Center of Gravity for total mass (for root node) if distance between center of gravity and node size (length of the quad) > threshold don't go through the node, calculate force between the two; otherwise check leaf nodes