Proposal for your Final Project

Handed out Tue, Feb 23. Due Thu, Mar 3 at 11:59pm. Submissions will be through the submit server (https://submit.cs.umd.edu/), either in the form of a pdf document (preferred) or as a Microsoft Word document. If you are on a team, only *one member* of the team need submit (but please include everyone's name). The final projects will be due late in the semester, perhaps the last week of classes. The exact due date will be announced later.

Overview: Give a short (roughly one page) high-level synopsis of your proposed game.

Team Members: List all the members of the team. (Working alone is fine. Personally, I prefer smaller teams, say 2–3 people, since it is easiest to balance the workload and avoid coordination issues.)

Game Title: Proposed title, which you can change later.

General Description: The game's general structure (e.g., FPS, RTS, etc.), number of players (one, two, or many), the game's general "look and feel" (3-d interactive, 2-d scrolling, turn based, etc.). What (if any) concrete games inspired your game? Please feel free to include illustrations or images.

It is not necessary to provide details at this point, and you are free to make changes in the future. (If the changes are significant, please keep me informed.)

Platform and Resources: On what system do you plan to implement/execute your game and what software tools (game engine, graphics, geometric modeling, physics, audio) do you envision needing? Have you (and your other team members) installed and tested the software systems that you plan to use?

Coordination: How will you and your teammates coordinate your work? (What sort of meetings do you plan? Where will the source files be maintained? Do you plan to use some form of shared file storage (Google docs) or revision control software (SVK or CVS)?

Practical Considerations: You are free to propose any general structure you like. The only pragmatic constraint is that it must be possible for you to present a short demonstration of the game to the class and to me and/or the Teaching Assistant periodically throughout the semester. Thus, if there is any special hardware needed, you will need to haul it onto campus and set it up a few times during the semester.

Tips:

Nothing is binding: You can make changes, even radical ones, throughout the semester. If you do, please keep me informed.

Previous projects: To get an idea of what is "doable," check out the videos of last semester's projects. (Of course, don't be influenced too strongly. A part of your grade is based on how innovative your project is.)

Start simple and extend: Start with a basic implementation so that you know that you have a basis to extend from. (Ambitious plans may fall apart near the end of the semester, leaving you with nothing to show.)

Do the hard things first: It is important to identify as early as possible in the development process any implementation issue that might hold up your progress. Try to determine such issues early and design a prototype to be sure that you achieve your minimum goals. You can add the "bells and whistles" later.

Do at least one thing well: Rather than doing a so-so job on many different elements, focus instead a single concept that will make your game stand out. (This might be something internal, such as a novel technical feature. If so, part of your final demo will involve an explanation of how you implemented this feature.)

Programming Assignment 1: Simple Unity Game

Handed out: Tue, Feb 9. **Due:** Part I is due Wed, Feb 17, 11:59:59pm and Part II is due Wed, Feb 24 at the same time. **Late policy:** up to 6 hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for each additional 24 hours. Submission instructions will be given later.

Overview: The goal of this assignment is to learn the basics of Unity by modifying the Roll-a-Ball tutorial on the Unity home page. There are two parts. The first is a very simple extension of the tutorial, and the second involves adding a new level with more complex behavior.

Part-I Requirements: Add the following modifications to the game given in the Unity Roll-a-Ball tutorial.

Variable pickups: In the online tutorial, there is a fixed number of pickups, and the game is won when you collect them all. Instead, there should be two public variables attached to some script in your program:

(1) the initial number of pickups (say, numberOfPickups) and

(2) the number of pickups to collect in order to win the game (say, pickupQuota).

This will allow the grader (from within the Unity editor) to alter their values and rerun your game. For example, you might add these public variables to your player-controller script. By setting the first to 15 and the second to 5, the game will start with a circle of 15 pickups, and you win the game as soon as you collect any 5 of them. (You may assume that pickupQuota $\leq$ numberOfPickups.)

Automatic pickup placement: In the tutorial, the pickups are placed by hand in the Unity editor. Instead, in one of your scripts you should automatically place the appropriate number (numberOfPickups) of pickups in a circle centered at the origin (where the player starts).

To do this, you can use the Instantiate function to create the pickups and Mathf.Cos and Mathf.Sin to determine the (x, z) coordinates of these points. We used a circle of radius 7.5 in our implementation.

Count-Remaining Text: Rather than showing a count of the number of pickups that have been collected so far, instead print a count of the number of pickups that remain to be picked up before winning the game. This number should be displayed in the upper-left corner. As in the tutorial, when the game is won an appropriate message is shown in the middle of the window.

Quit/Restart: After winning, provide the user the opportunity to either restart the game from the beginning or to quit. (For example, they might hit 'Q' to quit and any other key to restart.)

Freezing on termination: When the game has ended (either by winning or losing) everything should freeze. That is, the player ball should not move and the pickups should stop rotating.

You are allowed to modify these specifications (e.g., by altering the models, colors, and some aspects of game behavior), provided that your game demonstrates that you have mastered all the required elements listed above. For example, if your game uses mouse input rather than keyboard input, your player object should still be based on physics forces (as it is in the demo), and you should have some form of keyboard input (to demonstrate that you know how to do this).

Notes for Part I:

- The game can be restarted by reloading the initial scene. This can be achieved by invoking UnityEngine.SceneManagement.SceneManager.LoadScene(0).
- The game can be quit with the command Application.Quit().
  Note that quitting the game within the Unity simulator or a Web-based deployment does not do anything. At least, it didn't do anything with my implementation. However, if you produce an stand-alone executable, e.g., an ".exe" file on Windows, then quitting the game will terminate the program.
- Stopping the player from moving in physics mode is a bit tricky, since you need to eliminate all forces acting on the object. A simpler approach for causing a game object to stop moving is to remove it from the physics calculations by setting rigidbody.isKinematic = true, where rigibody is the rigid-body component of the player. This puts the player under the full control of the program, and if you do nothing to change its position, it will not move.

Part-II Requirements:

Multi-Level Structure: The game should have two levels, the level from Part I and a second level. For the grader's sake, there should be an easy way to bypass Level 1 and go directly to Level 2. It should also be possible to exit the game entirely at any time (say, by hitting the ESC key).

Different Environment: We modified the environment from Part I, creating a larger and more complex environment. First, the ground was reshaped to size 20×30, and we added gaps at the corners and at the centers of the two side walls. (There are effectively six walls.)

Player Can Jump: In addition to the existing horizontal motion along the ground, the player can also jump. We implemented this by adding a vertical force whenever the space key is hit. (This applies even if the player is already in the air.) If the player hits any of the other objects in the scene, it should bounce off of them. (see the Unity Bouncing Ball tutorial for information on how to do this using *Physic Material*.) If the player jumps outside of the ground area and falls so that the y-coordinate is negative, the game is immediately lost.

Power-Up State: The player can be in one of two states, *normal* and *powered-up*. We place a target above the center of the ground (a green disk in our implementation) that behaves like a trigger. If the player ball hits this trigger, it changes its appearance (we changed its color), and it transitions into the powered-up state for some fixed time period (we used 10 seconds). We used red for the normal state and green for the powered-up state.

The effect of pickup collisions depends on your state. When in the normal state, collisions with pickups are bad, and each collisions increases the number of pickups that you need to collect by one. Further, the pickup object does not vanish. In the powered-up state, each collision with a pickup causes it to vanish.

Pursuing/Evading Pickups: The pickups move. When the player is in the normal state the pickups chase after the player object. This is done by generating a force vector in the direction of the player. When the player is in the powered-up state, the pickups run away from the player.

If a pickup falls off the board as a result of this, then it dies, and the player is given credit for collecting the pickup. (In our implementation, the pickup does not die right away. It dies once its y-coordinate is smaller than -50.)

Jumping Beans: In our first implementation we found the pickup movement to be very boring. It makes their behavior much more interesting to have them jump periodically. To achieve this, at random moments (at the rate of roughly 1 per second) a sufficiently large vertical force is generated to cause it to jump a small distance in the air. (Unlike the player, pickups must be on or at least near the ground when they jump.)

Final Submission: (Final submission instructions will be forthcoming.)

Programming Style: We will be reading your code to see that you implemented everything in a reasonable manner. Although programming style is not an explicit part of your grade, we reserve the right to deduct points for programs that are so poorly documented or organized that the TA cannot figure out how your program is working.

Optional Elements for Extra Credit: You may add additional features to your game for the purposes of extra-credit points. (See the syllabus regarding extra-credit points.) Please explain any additional features are in your Readme.txt file. The number of points of extra-credit credit will be left to the discretion of the TA.

External Resources: If you make use of any external resources in your program (or things that you developed prior to this class), even if you modified them, you *must* credit them in your Readme.txt file. Failing to do so will be considered an act of plagiarism. If you are unsure, check with me.

Homework 1

Handed out Thu, Mar 3. Due at the start of class Tue, Mar 22. Late homeworks will not be accepted (without prior approval), so turn in whatever you have done.

This homework is intentionally structured in a similar manner as the First Midterm Exam. Of course, the problems on the exam will be different, and the length and technical difficulty of the Midterm will be different.

Problem 1. Short answer questions. Unless otherwise specified, explanations are not required, but may be provided for partial credit.

(a) Suppose that a Unity game object is declared to be *static* (by checking the "Static" checkbox in the editor). Which of the following optimizations can Unity perform as a result? (Indicate True or False for each.)

(i) Navigation computation and physics can be optimized (because the object's position is fixed).

(ii) Fewer method calls are needed, because the methods Update or FixedUpdate are not called on static objects.

(iii) Some global lighting computations can be precomputed.

(iv) Space is saves because all instantiations of a static object refer to the same (shared) game object.

(b) What does it mean when we say that the vector dot product (or generally any inner product) is *bilinear*? (Express your answer as one or more vector equalities.)

(c) Suppose that you want to represent a rotation of 60° about the rotation axis $\vec{u} = (1, 2, 2)$ in three-dimensional space. What is the quaternion that encodes this rotation? (Give your answer as a 4-element vector.) As a check of correctness, verify that the sum of squares of the elements of your vector is equal to 1.

(d) Consider the following two computational tasks that arise in animation processing:

Task I: Given the placement of a skeletal model in a scene and an assignment to its joint angles, determine the position of a point of the model (e.g., the tip of the index finger) relative to the scene's coordinate system.

Task II: Given the placement of a skeletal model in a scene and the desired position of a given point of the model (e.g., the tip of the index finger should be touching a light switch), determine how to set the joint angles to achieve this desired result.

(i) One of the above tasks is called *forward kinematics* and the other is called *inverse kinematics*. Which is which?

(ii) Which of these two tasks is computationally more challenging? Briefly justify your answer.

Problem 2. Everyone knows that zombies never turn their heads. If you are in front of a zombie, he will attack you, and otherwise he will ignore you. Suppose that there is a zombie in a

room. Let p_1 be a point on the top of the zombie's head, p_2 be a point at the tip of his right hand, and p_3 be a point at the tip of its left hand (as shown in Fig. 1(a)). The zombie is standing with his hands stretched out to the sides.

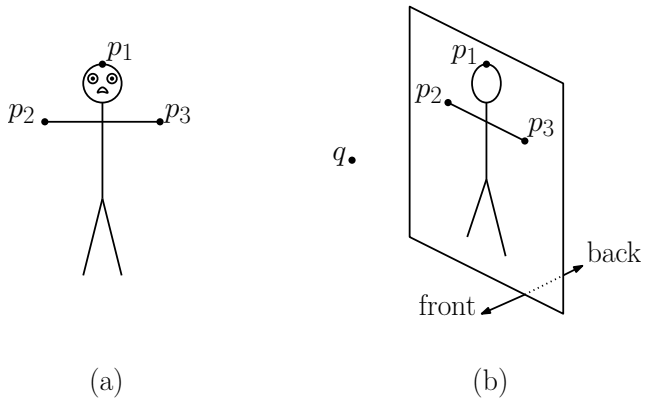


Figure 1: Problem 2.

Suppose that you are located at position q (see Fig. 1(b)). Present a geometric test to determine whether the zombie is facing you. In particular, if you imagine a plane passing through the zombie's body (as shown in the figure), determine whether q lies on the front side or back side of this plane. (You can ignore the possibility of q lying exactly on the plane.)

Problem 3. In class, we discussed a number of methods for determining whether certain pairs of colliders intersect (e.g., sphere-sphere, AABB-AABB, capsule-capsule). For this problem, you are given two colliders. Collider C_1 is an AABB (axis-aligned bounding box) in 3-dimensional space. It is defined by two points $p^- = (p_x^-, p_y^-, p_z^-)$ and $p^+ = (p_x^+, p_y^+, p_z^+)$. Collider C_2 is a sphere collider. It is defined by center point $c = (c_x, c_y, c_z)$ and radius r. (Recall that a point q lies within this collider if q lies within distance r of c, that is, $(q_x - c_x)^2 + (q_y - c_y)^2 + (q_z - c_z)^2 \leq r^2$.)

Present a simple procedure (in mathematical notation) that, given p^-, p^+, c, and r, determines whether these colliders intersect. (Hint: Start by considering how to compute the distance from c to its closest point on C_1. If c lies within C_1, this procedure should return zero.)

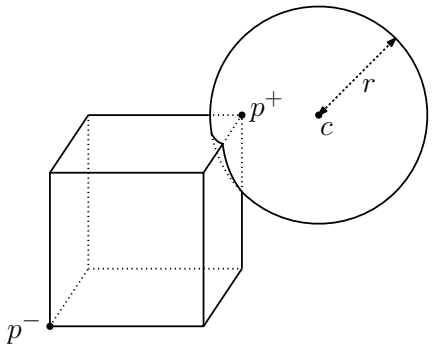


Figure 2: Problem 3.

Problem 4. Many games have aiming aids for weapons that shoot ballistic objects like bows that shoot arrows and catapults that throw rocks. In this problem we will consider how to implement one of these.

(a) Let's first consider a simple 2-dimensional scenario. The weapon is located at h meters above the ground along the y-axis, and is shooting the projectile with an initial velocity of $\vec{v}_0 = (v_{0,x}, v_{0,y})$ meters per second (see Fig. 3(a)). You may assume that both coordinates of $\vec{v}_0$ are positive. We want to determine the distance ℓ from the shooter where the projectile hits the ground.

Let $t = 0$ denote the time at which the object is shot. Also, let $g \approx 9.8 m/s^2$ denote the acceleration due to gravity (given in meters per seconds squared). We will ignore secondary issues like wind resistance.

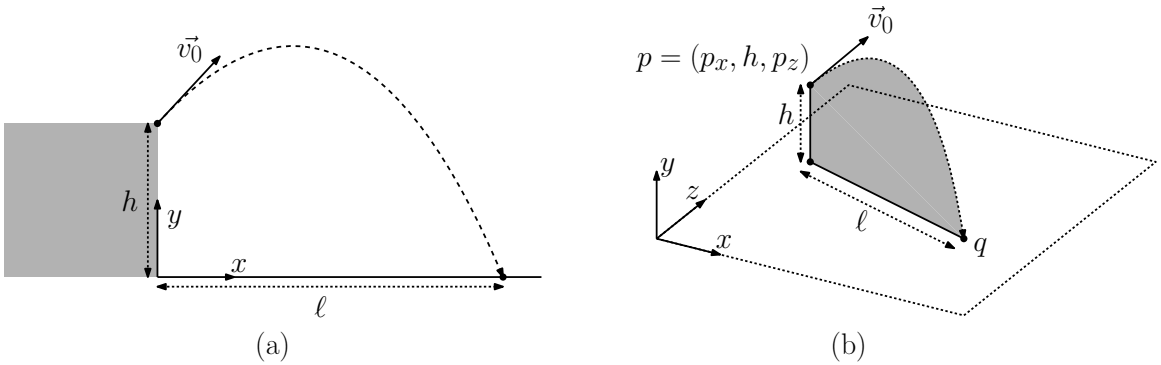


Figure 3: Problem 3.

After a quick visit to the Web, you find out that after t time units have elapsed, the position of the projectile is $p(t) = (x(t), y(t))$, where

$$x(t) \;=\; v_{0,x}t \quad \text{and} \quad y(t) \;=\; h + v_{0,y}t - \frac{1}{2}gt^2.$$

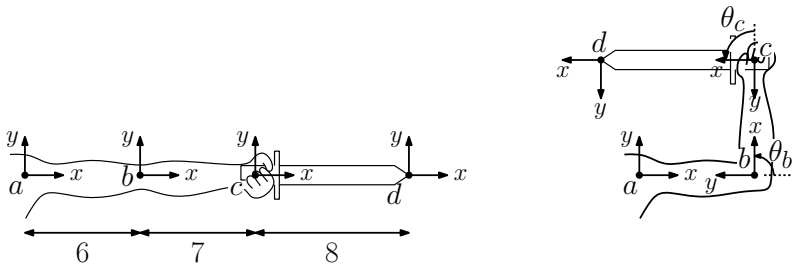
(i) Explain how to compute the time t at which the projectile hits the ground (that is, when $y(t) = 0$). Express your answer as a formula involving h, g, and the coordinates of $\vec{v}_0$. (Hint: The quadratic formula will be helpful.)

(ii) Letting t denote your answer to (i), what is the distance ℓ where the projectile hits the ground?

(b) For the 3-dimensional case, let us assume that the ground is the (x, z)-coordinate plane, and that the y-axis is directed upwards (see Fig. 3(b)). The weapon is located h meters above the ground at the point $p = (p_x, h, p_z)$. The initial velocity vector for the projectile is $\vec{v} = (v_x, v_y, v_z)$. As before, we assume that $v_y > 0$, so the projectile is being shot upwards. Show how to compute the coordinates of the point $q = (q_x, 0, q_z)$ where the projectile hits the ground.

Problem 5. Consider a skeletal model of an arm holding a sword in 2-dimensional space. Suppose that the bind pose is as shown in Fig. 4(a), with the arm and sword extending horizontally

to the right of the shoulder. The shoulder, elbow, hand, and tip of sword coordinate frames are called a, b, c, and d, respectively. It is 6 units from the shoulder to the elbow, 7 units from the elbow to the hand, and 8 units from the hand to the tip of the sword.



(a) (b)

Figure 4: Problem 5.

(a) Following the naming convention for the local pose transformations (given in Lecture 9) express the following local pose transformations a 3×3 homogeneous matrices.

(i) $T_{[c \leftarrow d]}$, which translates coordinates in the sword tip frame to the hand frame.

(ii) $T_{[b \leftarrow c]}$, which translates coordinates in the hand frame to the elbow frame.

(iii) $T_{[a \leftarrow b]}$, which translates coordinates in the elbow frame to the shoulder frame.

For example, the transformation $T_{[c \leftarrow d]}$ should transform the column vector denoting the tip of the sword relative to the tip-of-sword frame coordinate (as the origin) to its representation relative to the hand frame coordinates (as lying 8 units along the x-axis). That is,

$$T_{[c \leftarrow d]} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 8 \\ 0 \\ 1 \end{pmatrix}.$$

(b) Show that by multiplying these matrices together in the proper order, we obtain a matrix $T_{[a \leftarrow d]}$ that maps a point in the tip-of-sword frame to the shoulder frame. For example, because the tip lies 21 units to the right of the should, we have

$$T_{[a \leftarrow d]} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 21 \\ 0 \\ 1 \end{pmatrix}.$$

(c) Give the following inverse local pose transformations:

(i) $T_{[d \leftarrow c]}$, which translates coordinates in the hand frame to the sword tip frame.

(ii) $T_{[c \leftarrow b]}$, which translates coordinates in the elbow frame to the hand frame.

(iii) $T_{[b \leftarrow a]}$, which translates coordinates in the shoulder frame to the elbow frame.

(Hint: You can exploit the simple structure of the matrices in part (a) to avoid the need for general matrix inversion.)

(d) Suppose that we apply a rotation by angle θ_b about the elbow and θ_c about the hand. (These are both $90° = \pi/2$ in Fig. 4(b), but they can be any angle, positive or negative, in general.) Assume that $\mathrm{Rot}(\theta)$ denotes a 3×3 rotation matrix, that is

$$\mathrm{Rot}(\theta) \;=\; \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Let's assume that all points are represented in the shoulder frame. Following the example in Lecture 9, derive a matrix (which you may express as the product of a sequence of matrices) that maps a point representing the tip of the sword in the bind pose to its rotated position. For example, in the particular case where $\theta_b = \theta_c = 90°$, this would map the vector $(21, 0, 1)$ to $(-2, 7, 1)$. (Your answer should work for any values of θ_b and θ_c.)

Explain how you derived your answer.

Midterm Exam

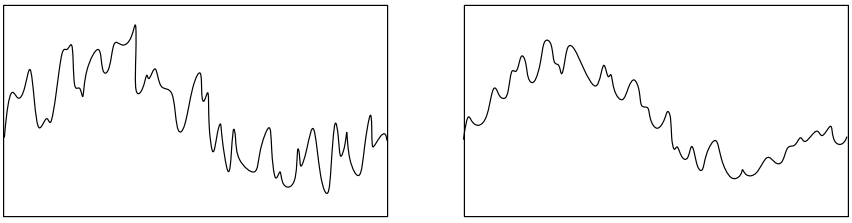
This exam is closed-book and closed-notes. You may use one sheet of notes (front and back). Write all answers in the exam booklet. You may use any algorithms or results given in class. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (25 points) Short answer questions.

(a) (8 points) Give two examples that might arise in a game implemented in Unity, one where you want a rigid-body to have a *collider* and one where you want a *trigger*.

(b) (5 points) You have three points p, q, and r in the plane. You want to compute a point that lies close to the center of this triangle (I don't care exactly where). Explain how to compute such a point using the operations of affine geometry.

(c) (4 points) You have just built a terrain using Perlin noise. You want to preserve the small bumps, but you would like them to be more subdued (see the figure below).



current desired

Which of the parameters that define Perlin noise would be best to alter to affect this change? (select one)

(i) Number of layers combined to form the final function
(ii) Lacunarity (frequency increase factor)
(iii) Peristence (amplitude decrease factor)

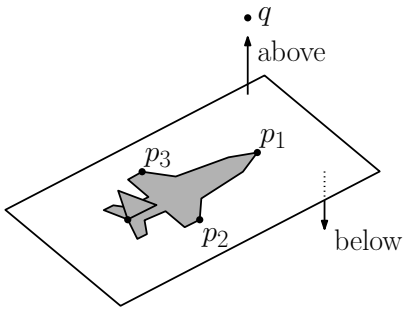
(d) (8 points) You have a long, thin object (e.g., an arrow) that can be oriented arbitrarily in space. Which of the following collider shapes would NOT be a good choice to represent this object (Select all the apply). Briefly *explain* your answers.

(i) Axis-aligned bounding box (AABB)
(ii) General (arbitrarily oriented) bounding box
(iii) Bounding sphere
(iv) Capsule

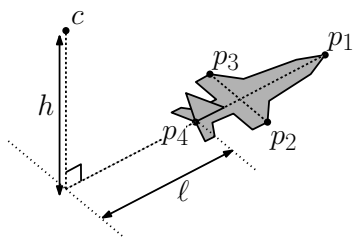
Problem 2. (25 points) Your flight-combat game involves aircraft that can fly at arbitrary angles. The notions of front/back, left/right, above/below are all defined *relative to the pilot* flying the aircraft. (For example, if the aircraft is flying upside-down then the ground is "above" the aircraft.)

Consider an aircraft that is defined by three points p_1, p_2, and p_3, where p_1 is the nose of the aircraft, p_2 is the tip of the right wing, and p_3 is the tip of the left wing. (See the figure below.)

(a) (5 points) The three points p_1, p_2, and p_3 define a plane in three-dimensional space. (See the figure below.) Given an arbitrary point q, explain how to determine whether q lies *strictly above* this plane, relative to the pilot's point of view.



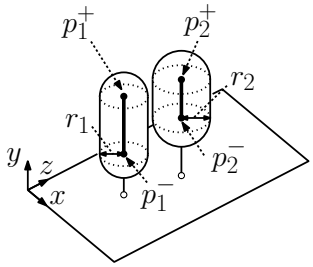
(a)



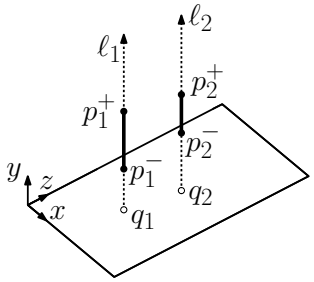
(b)

(b) (20 points) You want to define a camera that will follow the aircraft. Let p_4 be a point at the tail of the aircraft. The location c of the camera will be ℓ units behind the aircraft (by extending the line from p_1 through p_4 for ℓ units) and h units above the aircraft (along a line that is simultaneously orthogonal to $\overline{p_1p_4}$ and $\overline{p_2p_3}$). (See the figure below.) Explain how to compute c given $p_1, \ldots, p_4$ and ℓ and h. (Hint: Start by computing three unit vectors: $\vec{v}$ points forward, $\vec{r}$ points to the right, and $\vec{u}$ points up, all relative to the pilot.)

Problem 3. (25 points) Recall that a *capsule* is defined to be the set of points that lie within a given distance of a line segment, which is called the *central axis*. In class we discussed a relatively complex procedure for determining whether two capsule colliders intersect. This is *much easier* if the central axes of the two capsules are parallel to each other. The objective of this problem is to determine whether two capsule colliders C_1 and C_2 intersect, where the central axes of both are *vertically-aligned*.



(a)



(b)

For $i \in \{1, 2\}$, capsule collider C_i has radius r_i, the lower endpoint of the central axes is at $p_i^- = (x_i, y_i^-, z_i)$, and the upper endpoint at $p_i^+ = (x_i, y_i^+, z_i)$, where $y_i^+ > y_i^-$. The central axis is the line segment $s_i = \overline{p_i^- p_i^+}$. (See the figure (a).)

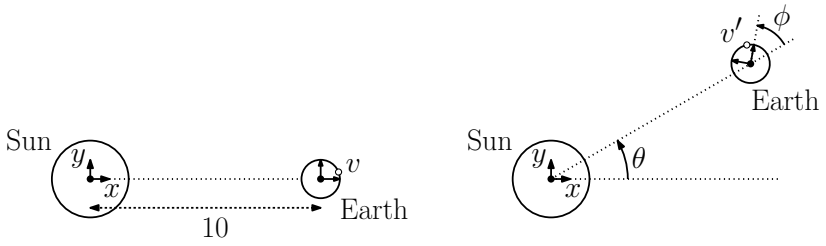
(a) (5 points) Let ℓ_1 and ℓ_2 denote the infinite vertical lines passing through s_1 and s_2, respectively. (See the figure (b).) Let q_1 and q_2 denote the points where these lines intersect the (x, z)-coordinate plane. What are the coordinates of q_1 and q_2? Given these coordinates, what is the (minimum) distance between ℓ_1 and ℓ_2?

(b) (15 points) Using the result from (a) and the y-coordinates of the endpoints of the central axes, explain how to compute the distance δ between the line segments s_1 and s_2? (Hint: You may want to split this into cases.)

(c) (5 points) Given δ from part (b) explain how to determine whether the two colliders intersect.

Problem 4. (25 points) NASA has asked you to help program a educational game for visualizing our solar-system. We'll simplify the problem by considering it in 2-dimensional space.

You are given a model of Earth, which is represented using a coordinate frame located at the Earth's center. You are given a model of the Sun, which is represented using a coordinate frame located at the Sun's center. (See the figure below (a).) The Earth rotates around its center and it revolves around the Sun. We want to compute the coordinates of a point v on Earth after this rotation and revolution. Let's assume a "bind pose" where the Earth is positioned 10 units away from the Sun along the x-axis.



(a) (b)

(a) (5 points) Let $v_{[e]}$ denote v's homogeneous coordinates relative to the Earth frame and let $v_{[s]}$ denote v's homogeneous coordinates relative to the sun's frame. Define $T_{[s \leftarrow e]}$ to be the affine transformation that maps a point in Earth-frame coordinates to the equivalent Sun-frame coordinates. Express $T_{[s \leftarrow e]}$ as a 3×3 matrix (assuming, as we usually do, that v is represented as a column vector).

(b) (5 points) Define $T_{[e \leftarrow s]}$ to be the inverse of $T_{[s \leftarrow e]}$. Express $T_{[e \leftarrow s]}$ as 3×3 matrix.

(c) (10 points) Let $\text{Rot}(\theta)$ be the affine transformation that rotates space by an angle of θ, expressed as a 3×3 matrix. Recall that

$$\text{Rot}(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Explain how to obtain a transformation that maps the point $v_{[e]}$ (in Earth coordinates) to its updated position $v'_{[s]}$ (in Sun coordinates) assuming a rotation of the earth by an angle ϕ and a revolution about the Sun by an angle of θ. (See the figure (b).) Express your answer as the product of a sequence of 3×3 matrices.

(d) (5 points) How would your answer to (c) change if instead the input was $v_{[s]}$ rather than $v_{[e]}$? (That is, v was given to you in Sun coordinates, not Earth coordinates.)

Programming Assignment 2: Animation and Navigation in Unity

Handed out: Thu, Apr 21. **Due:** Tue, May 3, 11:59pm. Late policy: up to 6 hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for each additional 24 hours. The submission procedure will be the same as in the first programming assignment (see below).

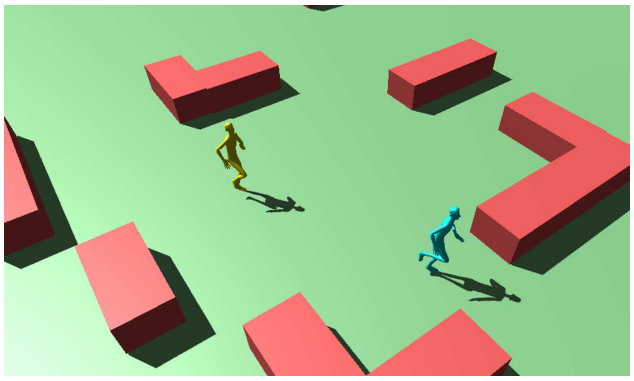
Overview: The goal of this assignment is to learn more about Unity, and in particular how to control the movement of characters through the use of animation controllers and navigation meshes. The assignment is based on the following Unity tutorial on animation and navigation:

Coupling Animation and Navigation:
http://docs.unity3d.com/Manual/nav-CouplingAnimationAndNavigation.html

That tutorial provides a complete Unity project (and explanations) for how to create a *2-dimensional blend tree* for creating blended animations, and how to use these in conjunction with Unity's NavMesh agent. The following links lead to Unity tutorials on related background topics:

- Animator Component: This is an introduction to how animations are controlled in Unity.
- Animator Controller: This controls the transitions between animations based on conditions that can be set in your scripts.
- Blend Trees: These are used to control how animations are smoothly blended between transitions.
- Navigation Meshes: This provides an introduction to how Navigation Meshes are constructed and processed in Unity.2

Start with the downloaded project (NavigationAnimation_45_and_50.zip) from the aforementioned Unity tutorial on Coupling Animation and Navigation. Read through the supporting documentation and test out the various optional features. This will produce a version of the final game in which there is a single moving agent (called "*The Dude*") that runs from its current location to anyplace in the scene (ground only) that you left-click on.



Your task is to create a second agent (called the *Follower*) that behaves in the following manner. Whenever The Dude comes within 5 units of the Follower, the Follower employs the NavMesh to plan a path whose destination is The Dude. The Follower moves more slowly than The Dude, and if The Dude ever gets more than 5 units away from the Follower, then the Follower stops. The Follower also stops whenever he is within 0.4 units of The Dude. When stopped, the Follower runs any idle animation (we used the same one as The Dude, namely, "take 001"). When running, we used a simple in-place jogging animation (called "Run_Spot".)

Note that these animations are not ideal, since the feet appear to slip on the ground. We will give extra credit if you provide a greater variety of animations (e.g., jumping or sneaking), blending between multiple animations, or better handling of general motion. Of course, any other features you add beyond the basic requirements will also be considered for extra-credit points.

Note that there is no objective of this game. If the Follower catches up with the Player, they just stand close to one another (and keep wiping their noses, which seems to be an artifact of the idle animation we are using).

Final Submission: We will use the same procedure that we did for the first programming assignment. Send an email to Ahmed with a link to a repository that contains a zip file with your submission. Also, see the Piazza posting by Ahmed ("Notes on Preparing the Submission Archive" from 2/16/2016) on how to prepare your submission files. Remember to include a ReadMe.txt file containing information about your submission and describing any external resources that you used (other than those mentioned in this handout or in class).

Programming Style: We will be reading your code to see that you implemented everything in a reasonable manner. Although programming style is not an explicit part of your grade, we reserve the right to deduct points for programs that are so poorly documented or organized that the TA cannot figure out how your program is working.

Optional Elements for Extra Credit: You may add additional features to your game for the purposes of extra-credit points. (See the syllabus regarding extra-credit points.) Please explain any additional features are in your Readme.txt file. The number of points of extra-credit credit will be left to the discretion of the TA.

External Resources: If you make use of any external resources in your program (or things that you developed prior to this class), even if you modified them, you *must* credit them in your Readme.txt file. Failing to do so will be considered an act of plagiarism. If you are unsure, check with me.

Homework 2

Handed out Tue, May 3. Due at the start of class Tue, May 10. Late homeworks will not be accepted (without prior approval), so turn in whatever you have done.

Problem 1. Short answer questions.

(a) Consider the design of a decision-making AI system based on *behavior trees*. Give an example of a task involving multiple decisions where a *sequence task* would be appropriate.

(b) Given the same scenario as (a), give an example of a task involving multiple decisions where a *selection task* would be appropriate.

(c) List one advantage and one disadvantage of the use of *potential-field navigation* as a means for computing paths in a game.

(d) When computing navigation meshes, we applied a step that simplified the polygonal region that modeled the walkable area. (That is, we approximated this region by eliminating vertices.) What was the principal reason for doing this simplification? What would be the danger of excessive simplification (removing too many vertices)?

Problem 2. Consider the triangle a and polygon b, shown in Fig. 1. (Triangle a's reference point is located at the origin, that is, $p_a = (0, 0)$ and b's reference point is at $p_b = (2, 4)$.)

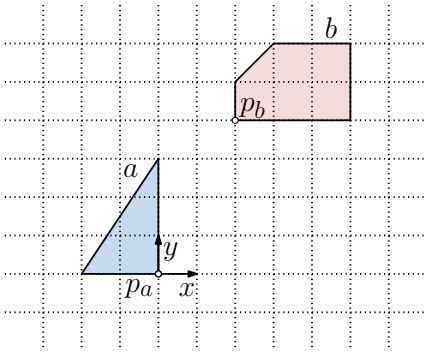


Figure 1: Problem 2.

The *configuration obstacle* of b with respect to a is the set of placements of a's reference point so that it overlaps b. Describe (draw clearly or explain with coordinates) the configuration obstacle of b with respect to a.

Problem 3. In this problem we consider the performance of Dijkstra's algorithm and A* search on the graph shown below (see Fig. 2), where the objective is to compute the shortest path from s to t. Each edge (u, v) is undirected and is labeled with its associated weight $w(u, v)$.

For A* search we need to make use of a heuristic. To save you from dealing with square roots, as we did in the example from class, we will use the L_1 (also called Manhattan or

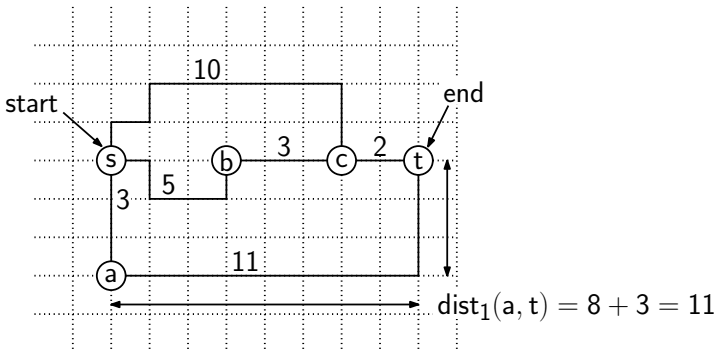


Figure 2: Problem 3.

checkerboard) distance between two points. It is defined to be the sum of the absolute values of the difference of the x and y coordinates of the points. For example, in the figure the L_1 distance between nodes a and t is $\text{dist}_1(a,t) = 8+3 = 11$. For A* search define the heuristic value for each node u to be L_1 distance from u to t. For example, $h(a) = 11$. (For this graph it is easy to verify that $h(\cdot)$ is an admissible heuristic.)

(a) Trace the execution of Dijkstra's algorithm on this graph. For each node indicate the following: (1) list the nodes in the order in which they are processed, (2) whenever a node is processed, indicate which of its neighbors have had their d-values modified, when the algorithm terminates (that is, when t is considered for processing), indicate what the final d-values are for all the nodes. If there are ties for which node is to be processed next, select the node that is earliest in alphabetical order. (As an example, see Lecture 19.)

(b) Trace the execution of A* Search on this graph. Provide the same information as in (a), but also provide the A* f-values for each node that is processed (recall that $f(u) = d[u] + h(u)$).

(c) Remark on the differences (if any) in the length of the path generated and the differences (if any) in the efficiency between these two algorithms.

Problem 4. Recall that in visibility-based pursuit-evasion games, you are given a domain and two agents, a pursuer p and an evader e. The pursuer selects a path π through the domain, and moves at constant speed along this path. The evader has knowledge of this path and can predict the exact location of the pursuer at any time. The evader can move at arbitrarily high speed. For a given pursuit path π, the evader *avoids detection* if it is possible for the evader to move in such a way that the pursuer never has a line of sight to the evader. If the pursuer can find a path π for which the evader cannot avoid detection, then the pursuer wins. If for every possible path chosen by the pursuer, the evader can avoid detection, then the evader wins

Consider the three domains shown in Fig. 3. For each, indicate whether the pursuer wins or the evader wins. If the pursuer wins, draw an example of a path π for which the evader cannot avoid detection. (For the fullest credit, your path should be reasonably short, and not involve unnecessary detours.) If the evader wins, give an intuitive explanation as to why

the pursuer cannot find such a path. (It is not necessary that your explanation is a rigorous proof. Explain where the "trouble spots" are within the domain.)

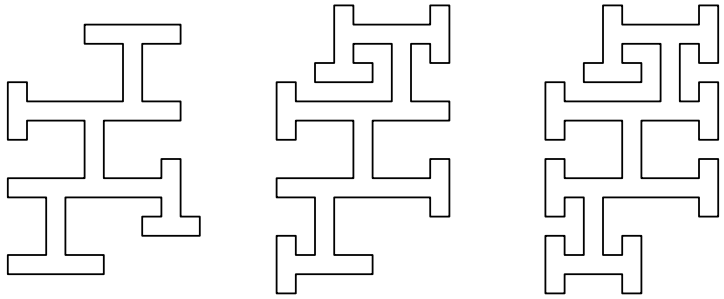


Figure 3: Problem 4.

Problem 5. In this problem we consider motion planning in a dynamic setting. The objective is to design a gamebot that can play a simplified version of the classic video game Galaxian.

You are given a *robot* that consists of a line segment of unit length that resides on the x-axis. The robot can move left or right (but not up or down) at arbitrary speeds. You are given two real values x^- and x^+, and the robot must remain entirely between these two values at all times (see Fig. 4). The robot's *reference point* is its left endpoint, and at time $t = 0$, the left endpoint is located at x^-. (You may assume that $x^+ > x^- + 1$.)

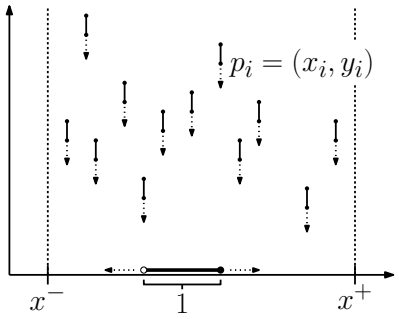


Figure 4: Problem 5.

You are also given a set of *missiles* in the form of n vertical line segments, each of length 0.2, that fall down from the sky at a speed of 2 units per second. Each of these vertical segments is specified by the coordinates of its lower endpoint at time $t = 0$. So, if $p_i = (x_i, y_i)$ is the starting position of the ith missile, then at time t its lower endpoint is located at $(x_i, y_i - 2t)$, and its upper endpoint is at $(x_i, y_i - 2t + 0.2)$. You may assume that $x^- \leq x_i \leq x^+$.

The question is whether it is possible for the robot to move in a manner to avoid all the missiles. We will explore an algorithm for solving this problem.

(a) A natural way to define the robot's configuration at any time is as a pair (t, x), where t is the current time, and x is the x-coordinate of the robot's left endpoint. Based on

this, what is the C-obstacle associated with a missile whose starting position is p_i (as defined above)? In other words, describe the set of robot configurations (t, x) such that the robot intersects this missile. (Explain/draw the exact shape of the C-obstacle, its dimensions, and its location in space. Don't just express it abstractly as a Minkowski sum.)

(b) Given the C-obstacles for all the missiles, under what circumstances is it possible to win the game? Express your answer by describing the properties of a path in configuration space, leading from the starting position to the end of the game. (I do not need a complete algorithm, just an explanation of what properties of the C-obstacle placements allow the game to be won.)

(c) Suppose you are told that the robot has a maximum speed σ (in units per second) that it can move at. How would you modify your answer to (b) to address this new limitation?