Proposal for your Final Project

Handed out Tue, Mar 3. Due Tue, Mar 10 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.

Who is Involved: 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.). 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.) Please feel free to include illustrations or images.

Platform and Resources: On what system to 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)?

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

Constraints: 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 Amit 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.

Advice:

Keep it small: Even if you use a game engine, it is hard to implement more than minimal content in the short time you have. Nonetheless, it is possible to create a fun and entertaining game with only very primitive elements.

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 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.)

Design in layers: Figure out what the bare minimum that you will need for a playable game, and make sure you have that running well before the final deadline. You can add on additional elements as time permits.

Programming Assignment 1: Simple Unity Project

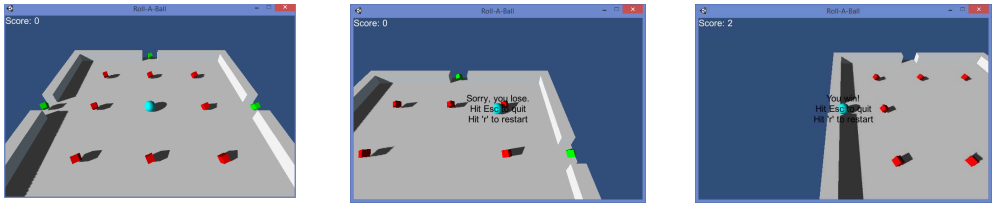
Handed out: Tue, Feb 10. **Due:** Wed, Feb 18, 11:59:59pm. Late policy: up to 6 hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for each additional 24 hours. Submit projects by uploading to the submit server.

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. See: `http://unity3d.com/learn/tutorials/modules`

Requirements: Make the following modifications to the game given in the tutorial:

Wall gaps: Place a gap in the middle of each of the four walls. If the rolling ball passes through this gap, it will fall off the board, resulting in the player losing the game.

Pickup types: There will be two types of *pickups* (the rotating cubes). Red pickups are bad, and hitting any of them causes the player to lose the game. Green pickups are good. In order to win the game the player must pick up any two of them. The green pickups are to be placed near the gaps in the walls (see Fig. 1(a)).



(a) (b) (c)

Figure 1: Screenshots.

Colors: Add materials to provide distinct colors to the player ball, the good pickups (green), and the bad pickups (red).

Text: Provide two forms of text. The score (number of good pickups collected) is shown in the upper-left corner. If the game is won or lost, a message is shown in the middle of the window (see Fig. 1(b) and (c)).

Quit/Restart: Add an end state to the game, which displays a message indicating whether the player has won or lost, and providing the opportunity to either restart the game from the beginning or to quit the game.

Freezing the player: When the game has ended (either by winning or losing) the player object should stop moving.

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 and Tips:

- Because the board is located on the $y = 0$ plane, you can detect whether the player ball has fallen off the board by checking that its y-coordinate (the vertical coordinate in Unity) is negative.
- Beware that the manner of generating text given in the tutorial (based on GUItext) is *not* supported by current versions of Unity (4.6 and higher). (It might remain as a legacy feature.) I would suggest using the current method, which is based on canvases and UI text. I'll say more about this in class, but further information can be found at `http://unity3d.com/learn/tutorials/modules/beginner/ui`.
- The game can be restarted by loading the initial scene, which can be achieved by invoking Application.LoadLevel(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. In my implementation I did not bother to stop the pickups from twirling around, but this would be an easy way to get some extra credit points.

Final Submission: Submissions of your final Unity project are to be made through the Department's submit server, https://submit.cs.umd.edu/. (The submit server is used only for uploading. All testing will be done by the TA.) Your submission will be in the form of a file archive. (You may use any standard archiving software, such as Winzip, WinRAR, or Unix tar and gzip. If you are unsure, check to see that the TA has your favorite archiver.) Compress the entire directory containing your project. Be sure that all assets used by your project are included.

Note that Unity submissions that contain executables can be extremely large because Unity adds many files that are common to all executables. To keep your submission to a reasonable size, please do *not* include executable files that were built as part of your submission. (For this project, if the size of your compressed submission is larger than 1MB, excluding any special resources you may have added for extra credit, you are probably including files that should not be there.)

In addition to the compressed Unity project file, also include a file called Readme.txt. This should contain everything the grader will need to know to run your program. (For example, if you added any additional features or know of any bugs, please include mention of them here.)

Trial Submission: If you are using any special elements, and you are unsure whether the TA will be able to handle them, you can check directly with him. Prior to the deadline, you can email him a partial submission, and he can tell you whether he had any difficulties executing it.

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.

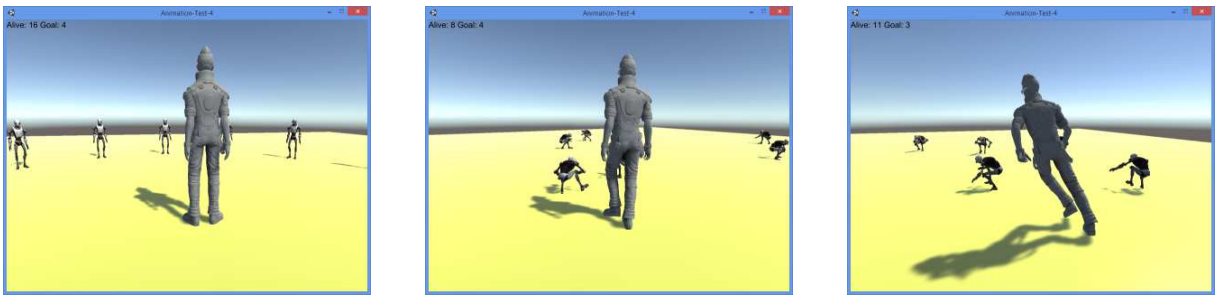
CMSC 425:Spring 2015 Dave Mount

Programming Assignment 2: Animation Control in Unity

Handed out: Tue, Mar 24. **Due:** Mon, Apr 6, 11:59:59pm. 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. Submit projects by uploading to the submit server.

Overview: The goal of this assignment is to learn more about Unity, and in particular how to control characters through the use of Unity's animation features. The project takes the form of a simple game involving an animated moving player and a collection of menacing robots (see Fig. 1(a)). Here are the principal elements:

Player: (roughly 50%) The game involves a humanoid player whose movements are controlled by the user input. When the user holds down the "W" or "↑" keys, the player walks forward (see Fig. 1(b)). If, in addition, the user holds down the "A" or "←" keys, the player walks forward while slowly turning to the left. Symmetrically, if the user additionally holds down the "D" or "→" keys, the player walks forward while slowly turning to the right. (There is no backing-up motion.)



(a) (b) (c)

Figure 1: Screen shots: (a) start, (b) walking, and (c) running.

The user can further modify any of these walking motions by holding down the left mouse button, in which case the walking behavior changes to running (see Fig. 1(c)). All of these actions should be implemented using a convincing form of animation. These animations should engage quickly (within a fraction of a second), should loop smoothly, and should transition smoothly from one state to the next.

Menacing Robots: (roughly 20%) Initially, the player is encircled by a collection (16, in our implementation) of small robot creatures. The robots are also animated. Each robot is in one of two states, *idle* or *alert*. When the player is not moving, all the robots are in their idle state. In our implementation, we used the same idle animation as for the player (standing and shifting slightly to the left and right).

Whenever the player is moving (either walking or running) the robots all transition to the alert state. In the alert state, the robots should move about independently of each

other. The exact nature of the motion is up to you. (In our implementation, we had the robots do a crouching walk, either left turning or right turning. The initial direction was chosen randomly, and at random moments (every few seconds), the walk switches from left turning to right turning. In this way, the robots have an overall appearance of moving independently, even though they are all using the same animator controller.) Whenever the player stops moving, the robots return to their idle (standing) state.

A robot's life is rather short and perilous. Whenever the player collides with a robot, the robot dies. Whenever two robots collide, they both die. Whenever a robot walks outside of the ground region, it dies. When a robot dies it simply disappears. (The definition of "collide" is rather lose. We used a simple capsule collider in our implementation, but you can use any method that looks reasonable.)

Camera: (roughly 5%) The player starts standing at the origin. The camera follows the player from a third-person perspective, always slightly behind the player.

Environment: (roughly 5%) The environment consists of a simple ground plane on which the player and robots stand. If the player moves off this ground plane, then the player instantly loses the game. (This condition need not be tested exactly. We simply checked whether the player's position coordinates lie outside the ground plane's extent.) There should also be a light source that casts shadows.

Scoring and Game Play: (roughly 20%) The game should keep track of two quantities, the number of robots that remain alive, and the number of robots that the player needs to hit before winning. When the player has hit a sufficient number of robots (four in our implementation), the player wins. (We have intentionally set the number low, so the grader doesn't have to spend many minutes playing your game in order to reach the winning state.) If all the robots have died (e.g., by colliding with each other or walking off the board) before the player reaches this goal, he loses.

As in the first programming assignment, when either ending state has been reached, an informative message should be printed, and the user should be given the option of quitting or restarting. (In some Unity deployments, quitting is not really an option, but this is okay.) When the end condition is reached, your program should freeze the display (until the user either restarts or quits). That is, the player and robots should not continue moving around.

General Requirements: As with all programming assignments, you are allowed some latitude in how you implement the game, as long as it is clear that you have mastered the necessary skills needed. In this project, the main elements are providing a complex mixture of animations (some controlled by the user and others controlled by the game logic), having the behavior of one game object affect the behaviors of others, monitoring interactions between different game objects, processing user inputs. If you are in doubt, please check with me.

Standard Models and Animations: For the sake of consistency, I would prefer that everyone use Unity's standard asset packages for models and animation. When creating your project, click on "Asset Packages" and select (at a minimum) "Characters" and "Space Robot Kyle." If you want to try something different, please check with me.

Animation: You should have at least two game elements (the Player and Robot) that exhibit complex animated behavior. Your program must include both Unity's *animator*

controller, in order to transition smoothly between different animations, and *animation blend-trees*, in order to interpolate between distinct animations (such as walking straight and walking while turning). In our case, the animator controller was used to transition between the idle, walking, and running states for the Player and between the idle and crouching states for the Robot. The blend-tree was used to blend between walking/running and turning animations. The particular Unity mechanisms needed for this assignment are described in two nice tutorials by Xenosmash Games:

`http://xenosmashgames.com/makehuman-mecanim-animations-unity-3d/`

`http://xenosmashgames.com/scripted-mecanim-animations-unity-3d/`

(Note that these tutorials are slightly out of date with respect to Unity 5, but I will provide some notes regarding the few needed modifications.)

I would prefer that you use the same inputs that we use in our implementation (which matches those of the tutorials). Again, if you would like to try something different, please check with me.

The relevant animations for our implementation were as follows. All can be found in Unity's standard assets folder (at least this is true for Unity 5):

Assets/Standard Assets/Caracters/ThirdPersonCharacter/Animation

Player and Robot idle: HumanoidIdle

Player walking (and turning): HumanoidWalk, HumanoidWalkLeft, and HumanoidWalkRight

Player running while turning: HumanoidRun, HumanoidRunLeft, and HumanoidRunRight

Robot crouching walk: HumanoidCrouchWalkLeft, and HumanoidCrouchWalkRight

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 and Tips: (Coming)

Final Submission: (Coming)

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.

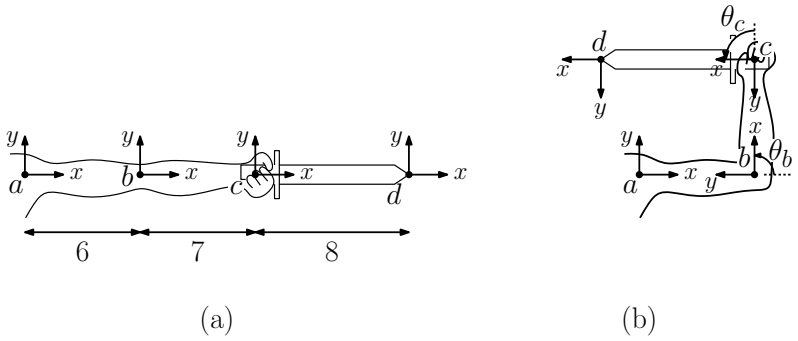
Optional Elements for Extra Credit: You may add additional features to your game for the purposes of extra-credit points. (See the syllabus regarding extra-credit points.) Please explain any additional features are in your Readme.txt file. The number of points of extra-credit credit will be left to the 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 Mon, Apr 27. Due at the start of class Thu, May 7. Late homeworks will not be accepted, so turn in whatever you have done.

Problem 1. Consider a skeletal model of an arm holding a sword in 2-dimensional space. Suppose that the bind pose is as shown in Fig. 1(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 1: Problem 1.

(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. 1(b), but they can be any angle, positive or negative, in general.) Assume that $\text{Rot}(\theta)$ denotes a 3×3 rotation matrix, that is

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

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.

Problem 2. 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 $w(u, v)$.

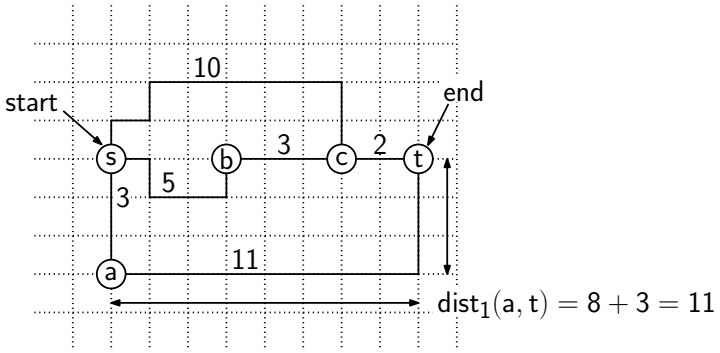


Figure 2: Problem 2.

For A* search we need to make use of a heuristic. To save you from dealing with square roots, we will use the L_1 (also called Manhattan or 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 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 18.)

(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 3. Consider the triangle a and rectangle b, shown in Fig. 3. (Triangle a's reference point is located at the origin, that is, $p_a = (0, 0)$ and b's reference point is at $p_b = (2, 3)$.)

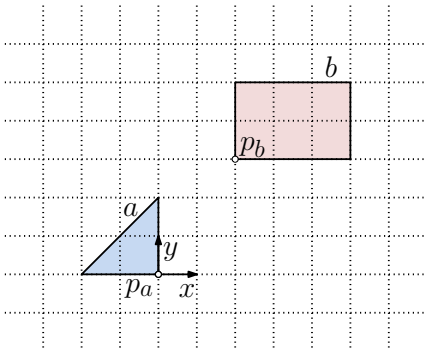


Figure 3: Problem 3.

(a) Recall that *placing* object a at a point q means translating a so that p_a coincides with q. The *configuration obstacle* of b with respect to a is the set of placements of a so that it overlaps b. Describe (draw clearly or explain with coordinates) the configuration obstacle of b with respect to a.

(b) Define $\text{VO}_{a|b}$ to be the set of all velocity vectors v such that, if triangle a moves with this velocity, it will intersect b at some time in the future, assuming that b is stationary. Describe $\text{VO}_{a|b}$ (again, draw clearly or explain with coordinates and slopes).

Problem 4. You are developing lag compensation software for an MMORPG. The following sequence of events are observed from the server's end:

- At time t_1, the server observes one player is at position $p_1 = (x_1, y_1)$ and another player is at position $p'_1 = (x'_1, y'_1)$ (see Fig. 4).
- At time $t_2 > t_1$, these two players are observed by the server to now be at positions $p_2 = (x_2, y_2)$ and $p'_2 = (x'_2, y'_2)$, respectively.
- At time $t_3 > t_2$, the first player executes an attack that hits all other players within a radius of r units of its current position (at time t_3). Unfortunately, the server does not know the exact position of the two player's at this point.

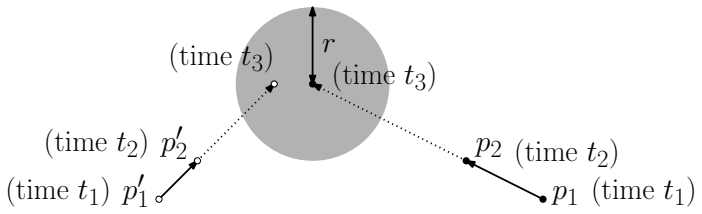


Figure 4: Problem 4.

Your task is to determine whether the attack from the first player will (likely) hit the other player. To do this, answer the following questions:

(a) Assuming that the first player continued from t_2 to time t_3 at the same average velocity that it had from t_1 to t_2, where would this player be located at time t_3? (Express your answer as a function of t_1, t_2, t_3 and the coordinates of p_1 and p_2.

(b) Assuming that the second player continued from t_2 to time t_3 at the same average velocity that it had from t_1 to t_2, where would this player be located at time t_3? (Express your answer as a function of t_1, t_2, t_3 and the coordinates of p'_1 and p'_2.

(c) Given your answers to (a) and (b) and the value of r, write an expression that determines whether the first player's attack hits the second player, under the assumption that both players move at the same average velocity. (Hint: If you expand everything out, the coefficients can get quite messy. It is probably a good idea to introduce some auxiliary variables along the way to store important values.)