A start-up guide to building multiplayer games using NetLogo

A start-up guide to building multiplayer games using NetLogo

Andrew Reid Bell (et al.?)
Department of Earth & Environment
Boston University
March, 2023

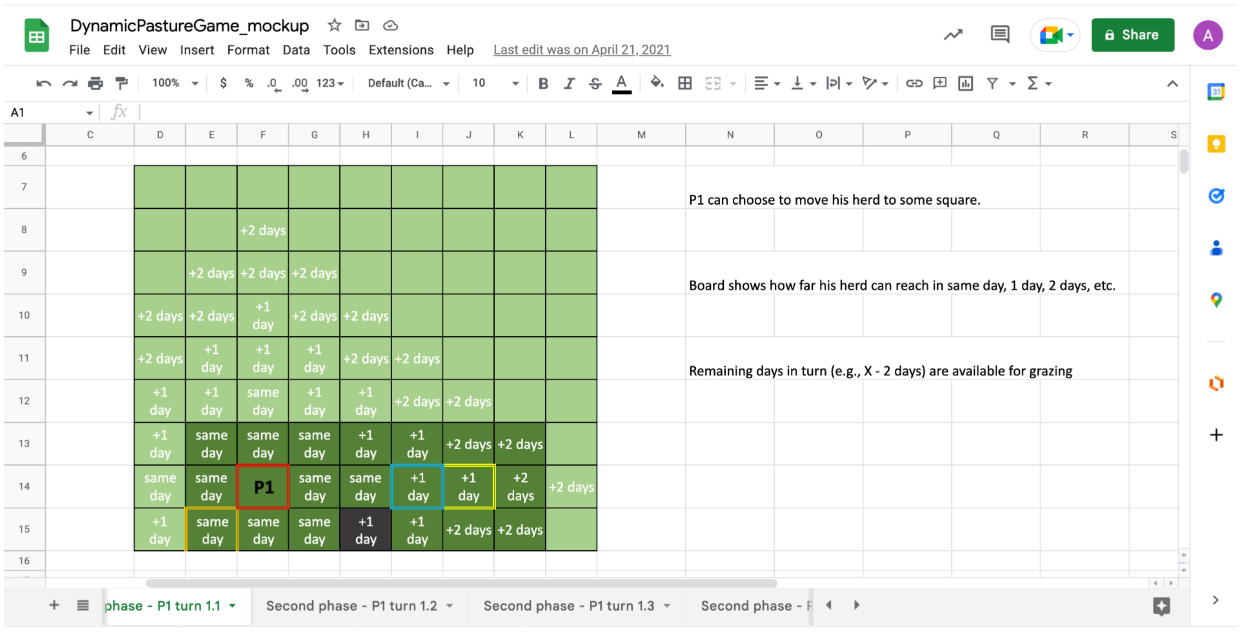
Thank you to XXXX for guidance in improving this document

Before you open NetLogo

This guide will focus more on the technical parts of operationalizing your game into the NetLogo platform. However, there is a lot of design work that comes before this. Importantly, before you dive into the things you *can* do with a modeling platform like NetLogo, it is worth investing time into identifying what you *should* do in order to achieve your objective - whether that is creating a learning experience for a group, or informing a specific research question. This up-front work will likely include:

- Identifying a dilemma
- Clear mapping (through expert consultation, focus-group discussion, or other participatory process) of how the dilemma plays out for people engaged in it, and what decisions they make
- Storyboarding your game into the different elements your instrument needs to map appropriately to the decisions and context you've learned about (e.g., Figure 1, showing one of the storyboard elements developed in expert consultation on effort trade-offs in pastoralism for the GreenReserve game)

We discuss many of the considerations you might have in our short courses and publications (see the end of this document). You also might consult other references, such as Angelino Viceisza's "Treating the field as a lab" or this recent synthesis of experiential learning in games by Thomas Falk, Wei Zhang, Ruth Meinzen-Dick and others.


DynamicPastureGame_mockup
File
Edit
View
Insert
Format
Data
Tools
Extensions
Help
Last edit was on April 21, 2021
Share
phase - P1 turn 1.1
Second phase - P1 turn 1.2
Second phase - P1 turn 1.3
Second phase -

	D	E	F	G	H	I	J	K	L
7									
8			+2 days						
9		+2 days	+2 days	+2 days					
10	+2 days	+2 days	+1 day	+2 days	+2 days				
11	+2 days	+1 day	+1 day	+1 day	+2 days	+2 days			
12	+1 day	+1 day	same day	+1 day	+1 day	+2 days	+2 days		
13	+1 day	same day	same day	same day	+1 day	+1 day	+2 days	+2 days	
14	same day	same day	P1	same day	same day	+1 day	+1 day	+2 days	+2 days
15	+1 day	same day	same day	same day	+1 day	+1 day	+2 days	+2 days	

P1 can choose to move his herd to some square.

Board shows how far his herd can reach in same day, 1 day, 2 days, etc.

Remaining days in turn (e.g., X - 2 days) are available for grazing

Figure 1: Early storyboard for 'GreenReserve,' one of the family of games produced using the approach to game design provided in this quick guide.

When you're ready for NetLogo

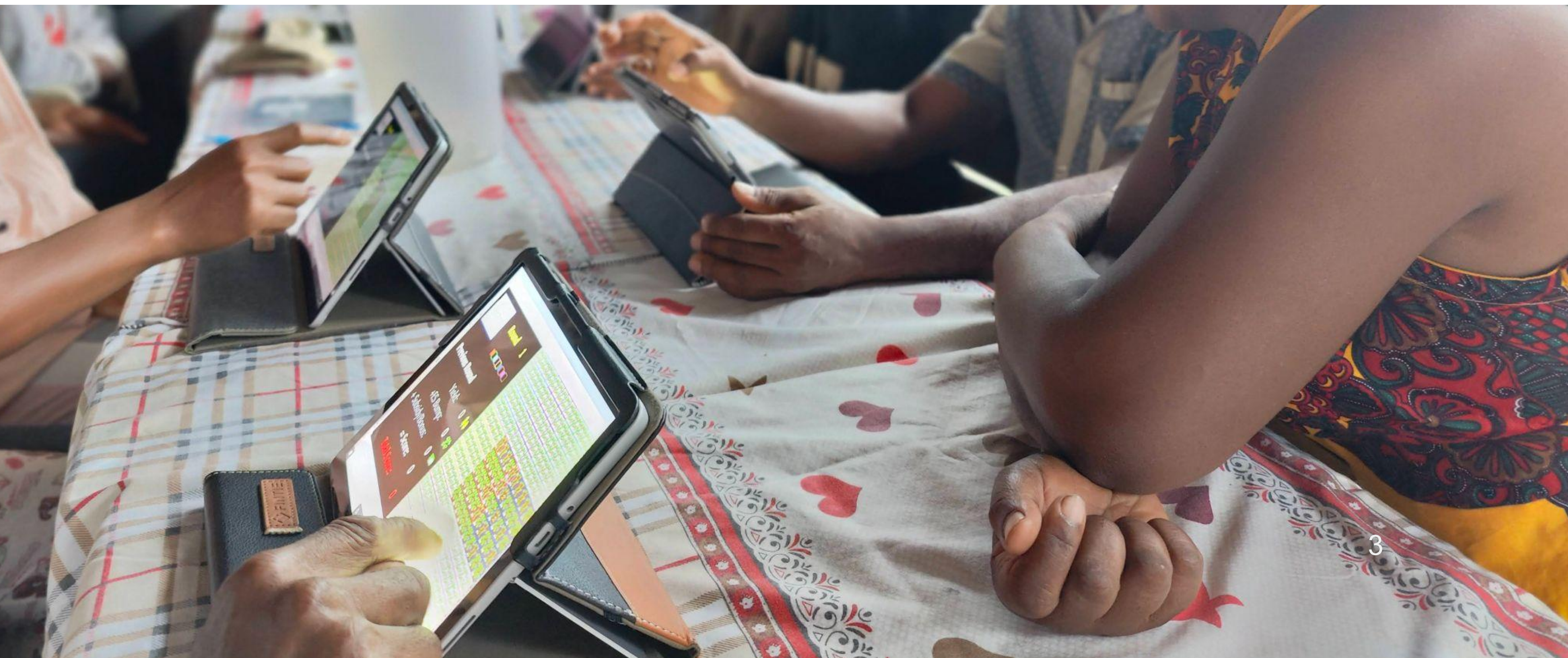
First, learn a bit about what NetLogo was built for - agent-based modeling - as it will help make sense of the tools we'll use. Next, learn what the pieces of our basic template mean, and practice a bit by adding on to the template to make some simple games.

This guide avoids being prescriptive wherever it can, and is very light on details about how to use NetLogo. Coding languages like NetLogo are like any other language - we use them to express concepts, and the best way to figure out how to express ourselves is to use them. NetLogo has great tools online - including a user guide, tutorials, and dictionary - for learning syntax, and like any other coding language there are helpful users all over the world providing examples and responding to queries online. Play with NetLogo, and use your Google or AI prompting skills to help work through challenges. I recommend working through the basic tutorials in NetLogo's documentation before looking through the following sections, as they'll assume some basic knowledge of how NetLogo works.

In this section, we'll outline some of the fundamental things that all multiplayer games require, and give examples of how we've managed them in NetLogo. Specifically, we'll show:

1) How to organize the NetLogo environment into a dynamic, spatial game world.
2) What Hubnet messages look like, and how to use them to have players interact with your game world.
3) How to make use of 'turtles' (NetLogo's mobile agents), 'patches' (NetLogo's stationary gridded agents), and imported bitmaps to create all of the visual elements of a game you might wish to have.

These basics cover the most technical aspects of game development in NetLogo. We'll walk through a few sample actions that build the environment above into things that are likely to be useful in games, but that's where we stop. Getting from there to the finished game product will come from your efforts at conceptualizing the game, trying things out yourself, and finding guidance online.



1 - Organizing the NetLogo environment into a game world.

If you're reading this, you've already done some background work with NetLogo. We won't cover the basics exhaustively here, but we will highlight a few things that are helpful in understanding how we turn NetLogo's capacity for participatory agent-based simulation into multiplayer games.

NetLogo is a decades-long project to build a language and environment for building agent-based models. It is built in Java, but is very carefully locked in to encourage ABMers (agent-based modelers, pronounced 'A-beamers') to think like ABMers. Some of the things you might want to do as a Java developer are not available to you. NetLogo programming means working within constraints, and becoming a better person for it.

In particular, NetLogo embeds the idea of contexts - three contexts, specifically. In NetLogo, you will code in observer, patch, and turtle contexts. The observer context sets up the model environment, and can make requests of patches and turtles. Patches can make changes to any part of the model environment, and make requests of other patches or of turtles - any code that patches execute is in a patch context. Same thing for turtles and turtle contexts. Importantly, patch context code can easily refer to the properties of the patch that will be executing it; turtle context code can refer to the turtle properties, etc. In some cases, code is context specific - for example, the observer uses 'create' functions to invoke new turtles, while patches and turtles use 'sprout' functions. Clever, right?

All code in NetLogo - whether for the observer, patches, or turtles to execute - is organized in procedures, and there are a few key procedures you are likely to need to set up and run a game.

You'll need a procedure that sets up all of the variables and visual elements you'll need within a game, that is executed at the start of each new game - clearing any elements from previous games, setting up the game 'board' and possibly a data output file, etc. - so that the game can then move forward through the actions taken on the hubnet clients. In our accompanying example, we've called this procedure start-game, and it can be viewed either by scrolling through the NetLogo code tab, or by selecting start-game from the 'Procedures' drop box at the top of the screen in the code tab.

Two further procedures that your game is almost certain to need, and which are discussed in more detail below, are the start-hubnet procedure - which launches the hubnet server, and which we also use to set any session-level variables that we would like preserved across games within the same session (such as the player names chosen by participants on each hubnet client) - and the listen procedure (which handles messages coming in from the hubnet clients).

The model view (what's happening in your game world) and controller (making things happen in your game world) are found in the 'Interface' tab in NetLogo. This includes the "View" (the 2D grid of patch agents in which all of your model processes will play out) as well as any other chart or text output viewers, and all user interface elements (buttons, sliders, switches) linked to global variables in your game. The accompanying template includes button elements to execute the three procedures above, as well as a switch element to select how grid squares are visualized in a simple clicking game. The hubnet client can include its own set of user interface elements (tailored for your game via the ‘Hubnet Client Editor’), though for the approach we develop here, we use only the model View.

And that is it - procedures to launch your game and handle messages, user interface elements to control game function, and choices about what to include in your hubnet client. In the next sections, we outline key hubnet functionality these procedures are likely to engage, and strategies for using NetLogo’s strengths to create dynamic, user-differentiated game experiences.

2 - Using Hubnet messaging to create an interactive, multiplayer environment

Hubnet is a server/client infrastructure component of NetLogo designed to allow participatory simulation - i.e., individuals controlling elements of a simulation. We use it in a related way, to allow individuals to play multiplayer games. The NetLogo models library includes a number of different simulations built using Hubnet, as well as a basic template of Hubnet functionality that is described fully in the Hubnet Authoring Guide within the NetLogo User Manual.

In our basic game template we use something similar, which we will outline in detail here. The key elements of the Hubnet system are i) the launching of a Hubnet server visible to all clients, ii) the opening of a 'listener' that receive messages from and sends messages to client machines, and iii) code to interpret and process the messages that come in, as well as set up and send new messages to clients.

2.1 Launching the Hubnet server

This one is easy. We'll use a procedure called 'start-hubnet' and assign it to a button in the interface. Within 'start-hubnet' we'll do a few things: 1) clear anything that might be left over from a previous game, 2) reset (or start up) the server, and 3) set anything else that we might want to maintain across all the games we play until we reset the server again (like player names, etc.). The procedure looks like this:

```
to start-hubnet

        clear-all        ;; clears everything left over
        hubnet-reset     ;; starts up the server

        ;; and below this line, set up anything we want to keep across all games
        set playerNames (list)
        ;; etc…

end
```

2.2 The listener procedure

This part is a bit more complicated, as it is the central engine for processing messages in your game, and so will encode some of the structure of your game in the way you write it. Hubnet clients send messages automatically when someone i) joins the server, ii) leaves the server, or iii) clicks something. These first two are simple to handle, while the third will scale with the complexity of your game.

Because this procedure very quickly grows long, I'll only show a simplified version of the listener procedure from our template here, to highlight the structure. Open the template in NetLogo alongside this guide to see the details.

```
to listen
    while [hubnet-message-waiting?] [
        set messageAddressed 0
        hubnet-fetch-message

        if(hubnet-enter-message? and messageAddressed = 0) [
            ;; CASE 1 - ENTER MESSAGES
            ifelse(member? hubnet-message-source playerNames) [
                ;;***CODE TO REFRESH AN EXISTING PLAYER'S STATUS
            ] [
                ;;***CODE TO ADD A NEW PLAYER TO THE GAME
            ]
        ] ;; end of CASE 1

        if(hubnet-exit-message? and messageAddressed = 0) [
            ;; CASE 2 - EXIT MESSAGES
            ;;***CODE TO TAKE NOTE OF A PLAYER EXITING THE GAME
        ] ;; end of CASE 2

        if(gameInProgress = 1 and messageAddressed = 0 and
            (member? hubnet-message-source playerNames)) [
            ;; CASE 3 - TAP MESSAGES

            if(hubnet-message-tag = "View") [
                ;;***CODE TO ADDRESS ANY SUB-CASES OF TAPPING
                THINGS IN THE VIEW
            ]
        ] ;; end of CASE 3

        if(gameInProgress = 1 and messageAddressed = 0 and hubnet-message-tag =
        "Mouse up") [
            ;;***CODE TO ADDRESS ANYTHING YOU'D WANT DONE
            DIFFERENTLY AT MOUSE UP***
        ]


    ] ;; end of while loop

end
```

NetLogo lacks a 'switch-case' structure like you might find in lower-level languages like Java, so we use a 'messageAddressed' flag to help organize the procedure - each case includes 'messageAddressed = 0' as one of the criteria in the if test, and will set messageAddressed to 1 if the code for that case is executed. No further cases will be executed for that message, and the loop will proceed to the next message.

The Hubnet authoring guide in the NetLogo documentation describes the hubnet built-in functions completely; I'll only highlight here the key functionality we use in our game listener:

hubnet-message-waiting? Returns true or false, whether there is a message in the queue to be read

hubnet-fetch-message Retrieves the next message to active memory, so that message properties can be accessed using other hubnet built-in functions, such as below

hubnet-enter-message? Returns true or false, whether the message is an enter message

hubnet-exit-message? Returns true or false, whether the message is an exit message

hubnet-message Returns the contents of the message. For click messages, the message contents are simply an [x y] pair of patch coordinates.

hubnet-message-source Returns the identity of the sender of the message

hubnet-message-tag Returns the nature of the message received from a mouse click. Mouse clicks always send two messages - the tag for the first one will indicate the type of interface element clicked on, while the tag for the second one will indicate "Mouse Up"

Using these basic built-in functions, the listener procedure watches for messages from the clients, fetches them one at a time, and triggers handling code as appropriate: Is a player returning to the game? Is it a new person? Did someone click somewhere? If so, what did they click on?

It is worth your time as a game developer to get comfortable with this code - modifying what is there and adding to it. Anything you add to your game that a player could click on - for which there is a specific response you wish triggered - will be pointed to in this procedure, and handled in a manner similar to what we introduce below.

2.3 Message handling
Most of the messages you'll have to code for will fit into a sub-case of what we've called our CASE 3 above - someone clicked something during the game. Each particular sub-case will take an if or an if-else statement, and look something like this:

```
if ("conditions describing the specific click sub-case") [

        ;;whatever code you want handled directly inside the listener procedure for this sub-case

        sub-case-XX-procedure ;;whatever code you port out to a named procedure

        set messageAddressed 1
]
```

where the procedure sub-case-XX-procedure contains code that you felt better placed in its own procedure.

What code should go in its own procedure and what shouldn't? This is your call as a developer. Many programmers will port code out to procedures if they know they will want to call it from different parts of the code, or if they wish to keep the visual flow of a larger procedure clear, or if they wish to be able to link more easily to the specific code they are porting (such as in NetLogo, where you can jump to any named procedure using the drop-down box at the top of the code screen). It's up to you. Figure out what works best for you. In this section, I only want to highlight a few bits of the hubnet built-in library that will be helpful for you as you handle these messages.
hubnet-send-message and hubnet-broadcast-message are built-in functions that send messages to specific players and all players, respectively.

hubnet-send-override and hubnet-clear-override are built-in functions that let you 'override' how a specific NetLogo object (a turtle or a patch, e.g.) appears on specific players' displays. These functions are invaluable to the interactive and differentiated experience of your game players - they allow you to have a patch or a turtle change color on one player's screen, for instance, to appear 'selected'; or for each player to see their own score (and not those of others) in a specific place on their screen.

To make clear how these tools can help, we'll turn to our last section, describing how to rope in NetLogo's agents to help you present a game experience to your participants.

3 - Using NetLogo's agents to express the features of your game dilemma

Ok, so now we've talked about how to set up procedures and how to get different players interacting via hubnet. We are at the crux of game development in NetLogo - how to get the game to look the way you want it to using only the tools at hand: turtles, patches, and imported images.

Turtles are mobile agents - they can have their own properties, they can move around the grid in any way you like, and they can have any appearance you can find (or draw) in the turtle shapes editor. They can also have a label - a number or string - with a catch: you can only have one font size for your whole world.

Patches are grid agents - for a view that is defined from 0 to X and 0 to Y, there is one patch agent at each point (x,y). Patches also have properties and can have labels - they just can't have shapes or move around.

We don't have drawing tools in NetLogo that you might find in other development environments - we can't just draw the lines, divisions, symbols we might like. We *can* use turtles shaped like lines or other symbols, and we can ask turtles to 'stamp' themselves on a place before moving on (or dying/disappearing). We can also import bitmap images using the 'bitmap' extension. This is a really useful approach in cases where we want to capture intricate designs (such as language tiles) that are impractical to represent using labels or turtle shapes. However, a drawback is that bitmap images and turtle stamps aren't objects we can interact with - once they're there, the only thing we can do is wipe the screen clear of them. Wherever it's possible to visualize what you want via an agent (patch or turtle), you will be able to tap into NetLogo's strengths.

NetLogo excels as an agent-based modeling platform by making it incredibly easy to call sets of agents and ask them to do things. All patches in a certain area? All turtles with the identifier 'tree'? All agents that are not currently visible? All very easy. By carefully identifying the different agents in your game (model), you can update what your players see and experience through their choices in the game. In the accompanying template, we use this property several ways just in the simple game example where clicking on an in-game patch leads it to change its color (or shape, depending on the settings).

3.1 - Example - changing a patch based on a click, and keeping count

First, in the procedure to-start-game, we include a block of code that designates a subset of the patches in the view as 'in game' and specifies what the player should see, depending on which visualization setting is selected at the game start:

```
ask patches [
        if-else (pxcor < 0) [
                set inGame? false
                set pcolor 0
        ][
                set inGame? true
                if-else (visualization_at_start = "colors") [
                        set pcolor red
                ] [
                        sprout-visuals 1 [set shape "fish"]
                ] ;; end if-else visualization
        ] ;; end if-else pxcor < 0
 ] ;; end ask patches
```

This block invokes the set of all patch agents using the built-in ‘patches’ and asks them to set their inGame? property based on whether they are above the x-axis or not. For those patches that are in the game, we also ask them to change how they appear - changing the patch color to red, if the visualization is by color, or by sprouting one of a breed of turtle we’ve named ‘visuals’ on top of them with a ‘fish’ shape. The breed declaration, as well as addition of the inGame? property to patches, is made at the top of the template code.

We then call the make-borders procedure, which uses agent ‘stamping’ to mark out boundaries between each patch:

```
to make-borders
        ask patches with [pxcor >= 0 and pycor < max-pycor] [
                sprout-borders 1 [set color border_color
                setxy pxcor pycor + 0.5
                set heading 90
                stamp die]
        ]
        ask patches with [pxcor >= 0 and pxcor < max-pxcor] [
                sprout-borders 1 [set color border_color
                setxy pxcor + 0.5 pycor
                set heading 0
                stamp die]
        ]
end
```

This procedure asks all of the patches within the in-game area to sprout ‘border’ agents whose shape is a simple line, have them move to the boundary with the next patch, make a ‘stamp’ of themselves, and die. We previously set in-game patches to have the inGame? property equal to 1, and could invoke them as a set by “patches with [inGame? = 1]” but since we want to do slightly different things for the lines at left/right than at top/bottom, we invoke two different subsets of patches: “patches with [pxcor >= 0 and pycor < max-pycor]” and “patches with [pxcor >= 0 and pxcor < max-pxcor].” Importantly, we very easily and temporarily invoke these specific sets, without needing to create and store lists of agents in advance. Importantly as well, because we’ve removed the agents and have only their stamps left, we can’t interact with them or modify them further, except by wiping the screen clean. If we wanted to regularly make changes to this grid of lines, we might consider keeping the agents in place and engaging them by calling “borders” or “borders with [(some property of the borders)].”

The last piece of this process - updating what the patches look like in response to a click - is handled by the update-patch-state procedure, that is called within the listener procedure when the specific case of a “mouse-click within the view” message is received from a hubnet client. Specifically, it is called as “ask patch xPatch yPatch [update-patch-state currentPlayer].” That is, it is called by *asking* the patch that was clicked to do something, given additional information of which player clicked. This means that update-patch-state is called by a patch (not the observer) and is in a patch context - all of the patches properties (like color) are automatically in scope.

```
to update-patch-state [currentPlayer]
        ;;mark off whether we are doing colors or shapes with a shorter flag visType.
        let visType 0
        if visualization_at_start = "colors" [set visType 1]

        ;;get a flag for the state of the cell that will be a 0 or a 1.  In colors, red = 1; in shapes,
        fish = 1
        let cellState 0
        if-else (visType = 1)
                [if (pcolor = red) [set cellState 1] ]
                [if ([shape] of one-of visuals-here = "fish") [set cellState 1] ]

        ;;now make changes - from 0 to 1, or 1 to 0.
        let currentCount item (currentPlayer - 1) playerCounts
        if-else (cellState = 1) [
                if-else visType = 1 [set pcolor green][ask visuals-here [set shape "airplane"] ]
                set currentCount (currentCount + 1)
        ] [
                if-else (visType = 1) [set pcolor red][ask visuals-here [set shape "fish"] ]
                set currentCount (currentCount - 1)
        ]
        set playerCounts replace-item (currentPlayer - 1) playerCounts (currentCount)
        ask counters with [identity = "greenPatches"] [
                hubnet-send-override (item (position currentPlayer playerPosition) playerNames)
                self "label" [currentCount]
        ]
end
```

Patches (and turtles) can access and modify any global variable, just like the observer. This code also references ‘pcolor’ directly, which will refer to the pcolor of the patch that is calling this procedure. If the visualization is set to ‘shapes,’ then the patch will have a visuals agent sitting on it, which it accesses by calling the agent set ‘visuals-here’ - a built-in function that returns the set of all agents of the breed specified that are on the current patch (we happen to know there will be exactly one of them). Properties of other patches and agents called by this patch in this procedure can be reached by using ‘of’ - e.g., “[shape] of one-of visuals-here” and modified by asking the patch or agent to change them - e.g., “ask visuals-here [set shape "fish"].”

The last piece of this procedure points to one particular agent - of a breed we’ve called ‘counters’ and with an identifier of “greenPatches” - and asks it to change how it appears on the device of the player who clicked (and whose device sent the message through hubnet). We are storing a few things in lists - all of the player names (which identify the hubnet client to send messages back to), and the running count of the net number of patches changed to state 1 by each of those players. We use hubnet-send-override to ask the counter agent identified by “greenPatches” to change the number shown in its label to player X, to be the current running count for player X. This mechanism - overriding what an agent looks like on a particular client’s screen, or clearing it with hubnet-clear-override - allows us to customize appearances and allow information to be private to particular players. This enables mechanisms such as selections and choices that are not public to other players until later in the game (if at all).

3.2 Using imported images

Where the thing you want to show is just too complicated to be drawn with turtle shapes, the bitmap extension can help. As an example, we have used them in field settings where we would like the language of text elements to be changed via an in-game setting - rather than use actual buttons whose labels must be edited individually, we create virtual buttons using bitmaps whose appearance we can control completely by printing a bitmap image in the view.

A few drawbacks to bitmaps:

1) They aren’t objects you can interact with, so that once they’re drawn the only option you have is to clear / redraw the view if you wish to change them.
2) Bitmaps don’t reliably update automatically in Hubnet clients. This is an issue that may be corrected in future, but at present, the only reliable way that we have found to have a bitmap that has been drawn in the view appear in the hubnet clients, is to have one client exit and re-enter the game. For static images, this is a minor inconvenience; for any dynamic use of bitmaps (drawing, clearing, redrawing), it is a greater issue.

The above issues aside, bitmaps are great tools to customize the appearance of your game. In the accompanying template, we add the bitmap extension at the top of the NetLogo code, and then call functions from this extension in the setup-panel procedure, invoked by the button of the same name.

The bitmap extension draws at locations specified in pixels, whereas commands for turtles and patches specify locations in patches. You may wish to write quick conversion procedures between pixel and patch locations; I also like to express locations as 'fractions' (of total height or width, e.g.) as these will be robust to any resizing of the view. In the accompanying template, we specify a rectangular area as a four-element list of [x-minimum y-minimum width height], and use helper functions to convert this list across fraction, patch, and pixel spaces:

```
to-report convertFracPix [fracLoc] ;;converting fractional measures to pixel measures
        let pixLoc (list ((item 0 fracLoc) * x_panel_pixels) ((item 1 fracLoc) * y_pixels) ((item 2
        fracLoc) * x_panel_pixels) ((item 3 fracLoc) * y_pixels))
        report pixLoc
end

to-report convertPixPatch [pixLoc] ;;converting pixel measures to patch measures
        let patchLoc (list (min-pxcor - 0.5 + (item 0 pixLoc) / pixelsPerPatch) (max-pycor + 0.5 -
        (item 1 pixLoc) / pixelsPerPatch) ((item 2 pixLoc) / pixelsPerPatch) ((item 3 pixLoc) /
        pixelsPerPatch))
        report patchLoc
end
```

With these helper functions, we designate a space for our 'Confirm' button within the setup-panel procedure and place the image using two functions from the bitmap extension - bitmap:scaled (to rescale our image to the size of the box we've specified) and bitmap:copy-to-drawing (to paint it in the view):

```
set confirmFracLoc (list 0.55 0.025 (0.625 * 1.2) (0.208 * 0.8))
set confirmPixLoc convertFracPix confirmFracLoc
set confirmPatchLoc convertPixPatch confirmPixLoc
bitmap:copy-to-drawing (bitmap:scaled confirmTile (item 2 confirmPixLoc) (item 3
confirmPixLoc)) (item 0 confirmPixLoc) (item 1 confirmPixLoc)
```

We transform this painted image into a virtual button with a simple function to evaluate whether the location in a hubnet click message falls within the area of the button, implemented in the accompanying template as clicked-area:

```
to-report clicked-area [ currentPixLoc ]

        ;; inputs are boundaries in PIXEL SPACE
        let xPixel ((item 0 hubnet-message) - min-pxcor + 0.5) * patch-size
        let yPixel (max-pycor + 0.5 - (item 1 hubnet-message)) * patch-size
        let xPixMin item 0 currentPixLoc
        let xPixMax item 0 currentPixLoc + item 2 currentPixLoc
        let yPixMin item 1 currentPixLoc
        let yPixMax item 1 currentPixLoc + item 3 currentPixLoc
        ifelse xPixel > xPixMin and xPixel < xPixMax and yPixel > yPixMin and yPixel < yPixMax
                [report true]
                [report false]
end
```

In the accompanying template, Case 3.1 within the listener function uses clicked-area to check whether the player clicked the confirm button, and runs appropriate code if clicked-area returns true.

And that's it - these are examples of all of the basic tools you can use to leverage NetLogo's participatory simulation capacity into dynamic, multiplayer games. The 'Info' tab in the accompanying template includes a few guiding suggestions of next steps - how you could take the simple template and turn it into a resource mining derby.

So now what?

When you want to take this further

Ok, so where to go next?

We used this guide to outline a few of the basic mechanisms available in NetLogo and Hubnet for participatory simulations, that you can harness to develop interactive, dynamic games that playable robustly in any environment.

We didn't spend a lot of space talking about how to make good games, or how to use NetLogo, and these are both skills to build in tandem if you would like to develop and refine a good game instrument.

If you have a good game idea and are comfortable with NetLogo code, get coding! We have a growing family of hubnet-based games that could be launch points for your related game idea. You may find it convenient to work from these as a base to develop your idea, or reach out to us (bellar@bu.edu) - we may be able to adapt the games easily to fit your purpose.

If you found this guide or any of our games useful in developing your game, please consider citing them in your work; prior publications with this framework include:

- *Bell, A. R., Rakotonarivo, O. S., Bhargava, A., Duthie, A. B., Zhang, W., Sargent, R., Lewis, A. R., & Kipchumba, A. (2023). Financial incentives often fail to reconcile agricultural productivity and pro-conservation behavior. Communications Earth and Environment, 4(2023), 27. https://doi.org/10.1038/s43247-023-00689-6*
- *Sargent, R., Rakotonarivo, O. S., Rushton, S. P., Cascio, B. J., Grau, A., Bell, A. R., Bunnefeld, N., Dickman, A., & Pfeifer, M. (2022). An experimental game to examine pastoralists' preferences for human–lion coexistence strategies. People and Nature, June, 1–16. https://doi.org/10.1002/pan3.10393*
- *Rakotonarivo, S. O., Bell, A. R., Abernethy, K., Minderman, J., Bradley Duthie, A., Redpath, S., Keane, A., Travers, H., Bourgeois, S., Moukagni, L. L., Cusack, J. J., Jones, I. L., Pozo, R. A., & Bunnefeld, N. (2021). The role of incentive-based instruments and social equity in conservation conflict interventions. Ecology and Society, 26(2). https://doi.org/10.5751/ES-12306-260208*
- *Rakotonarivo, O. S., Bell, A., Dillon, B., Duthie, A. B., Kipchumba, A., Rasolofoson, R. A., Razafimanahaka, J., & Bunnefeld, N. (2021). Experimental Evidence on the Impact of Payments and Property Rights on Forest User Decisions. Frontiers in Conservation Science, 2(July), 1–16. https://doi.org/10.3389/fcosc.2021.661987*
- *Rakotonarivo, O. S., Jones, I. L., Bell, A., Duthie, A. B., Cusack, J., Minderman, J., Hogan, J., Hodgson, I., & Bunnefeld, N. (2020). Experimental evidence for conservation conflict interventions: The importance of financial payments, community trust and equity attitudes. People and Nature, August, 1–14. https://doi.org/10.1002/pan3.10155*
- *Bell, A., & Zhang, W. (2016). Payments discourage coordination in ecosystem services provision: evidence from behavioral experiments in Southeast Asia. Environmental Research Letters, 11, 114024. https://doi.org/10.1088/1748-9326/11/11/114024*
- *Bell, A., Zhang, W., & Nou, K. (2016). Pesticide use and cooperative management of natural enemy habitat in a framed field experiment. Agricultural Systems, 143, 1–13. https://doi.org/10.1016/j.agsy.2015.11.012*