Documentation Tangram Experiment

Content

1 Experiment Description

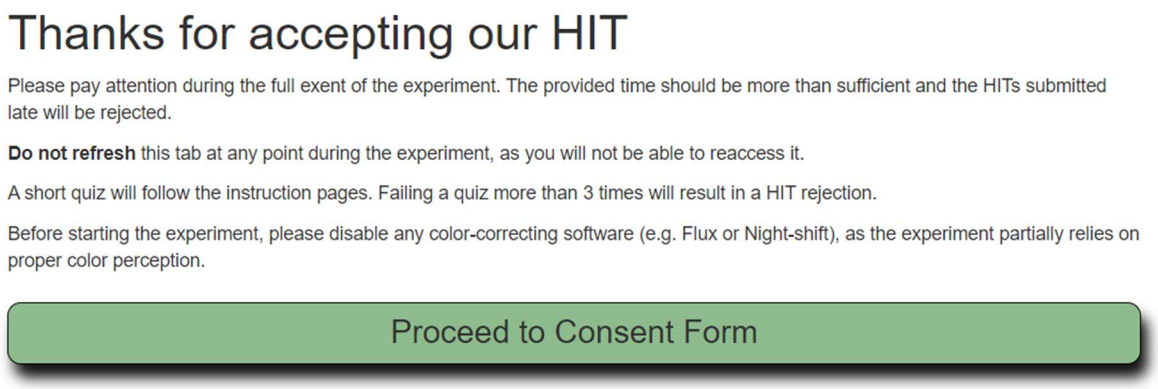
2 Experiment from the participant perspective

2.1 Start of the experiment

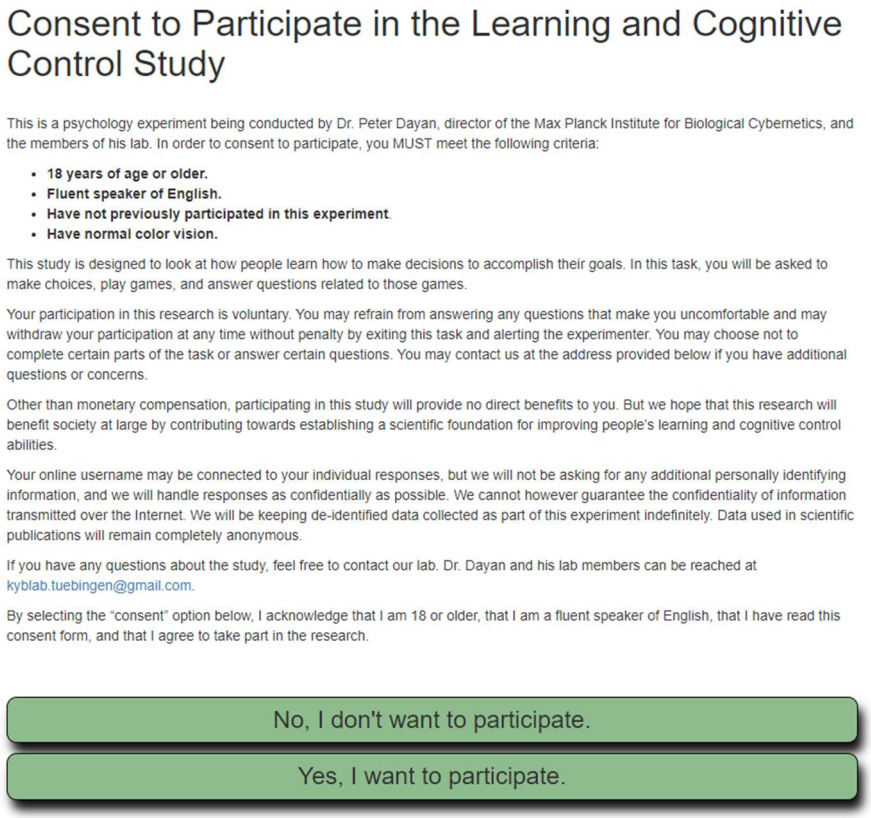
The online experiment consists of multiple sub-pages and different tasks which are sequentially displayed to the participant. A usually consists of multiple trials and if all trials are finished, the application will load the next task or the end of the experiment.

Not every experiment consists of all the different tasks and can also differ in the number of trials for each task. Which tasks that are part of an experiment, the number of trials and other options can be set in a separate configuration file. For further information see Chapter 3.

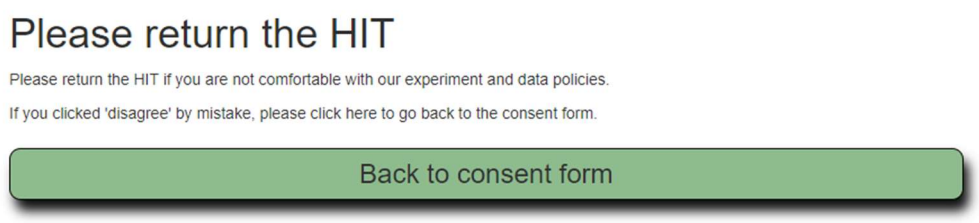
The online experiments are usually done via Prolific. If a participant starts the experiment, he or she will be redirected to the website and the welcome page will be shown:



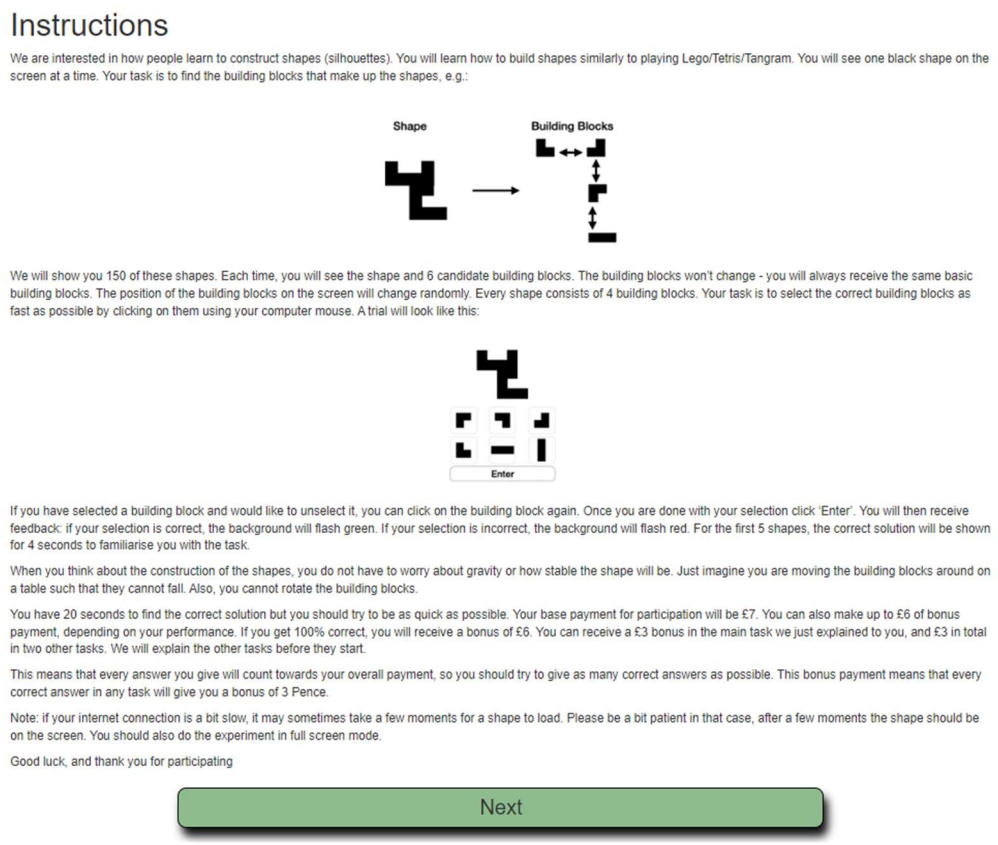
After clicking on the Proceed Button, the consent form to participate is shown which the participant can either accept or decline.



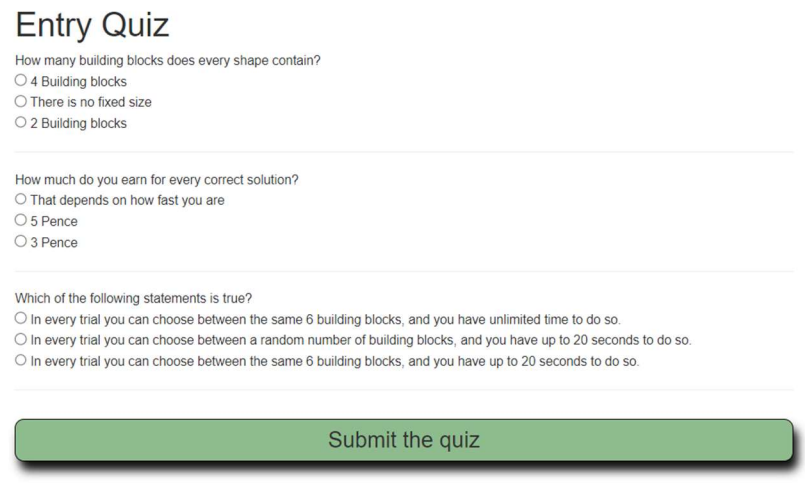
When declining the consent, an information text is shown, given the opportunity to return to the consent page.



When accepting the conditions, the participant is redirected to the instruction page, where basic information about the task, time limit and payment are provided:



On the next page the participant has to fill out a multiple-choice quiz to make sure he or she read the instructions carefully.



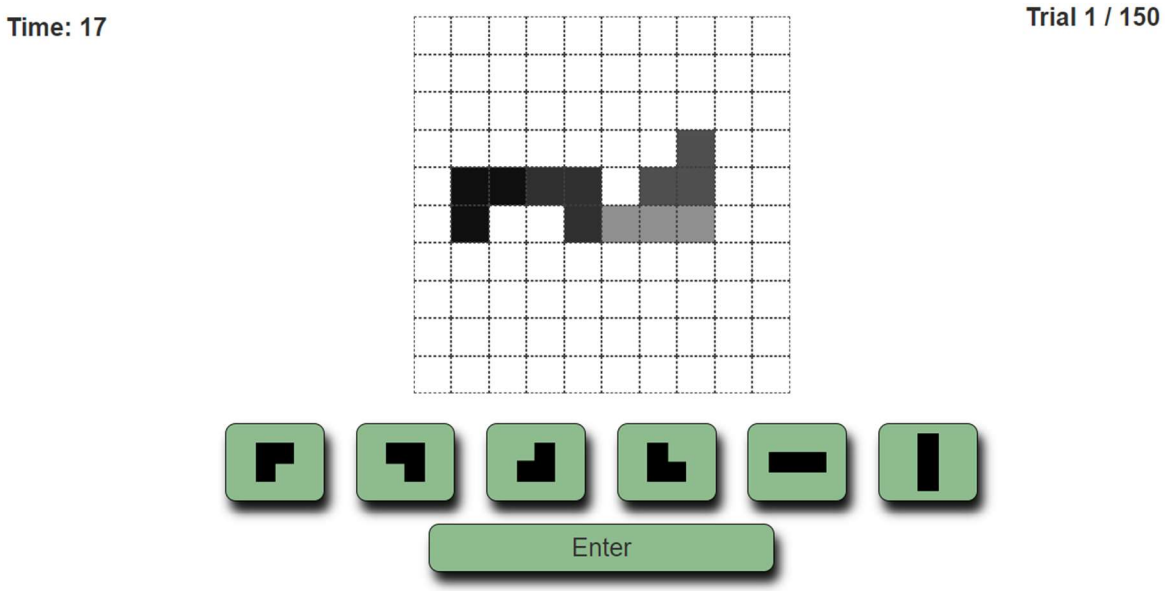
After submitting the answers of the quiz are evaluated. Only if all answers are correct, the participant may proceed in the experiment. The participant can fail the quiz up to three times (the questions and possible answers stay the same). After failing for a third time, the user is excluded from the experiment.

(The correct answers for the current quiz as shown in the screenshot above are 1,3,3)

Up to this point the displayed pages will stay the same for every experiment. In the following, all the different tasks that can be part of the experiment are presented. Note that the presence and order of certain tasks can change and it is also possible to repeat certain patterns of tasks multiple times during the experiment (For further information see Chapter 3).

2.2 Main Task

This is the main task of the experiment and is usually part of every experiment configuration.



In every trial of the task, a silhouette a certain silhouette is shown. The shape consist of a certain number (here four) of primitive building blocks. There are six different building blocks. The possible building blocks are displayed in the green buttons below the silhouette.

The participant has to click on the building blocks that are contained in the silhouette and submit the selection by pressing enter. Note that the outline of the different building blocks (displayed in different grey colors in the screenshot above) are only displayed in the first 5 trials of the experiment and only for a few seconds to demonstrate how the task works.

Every trial has to be solved in a certain time limit. How many seconds are left for the current trial is displayed in the upper left corner. The current trial and the number of total trials of this task are displayed in the top right-hand corner.

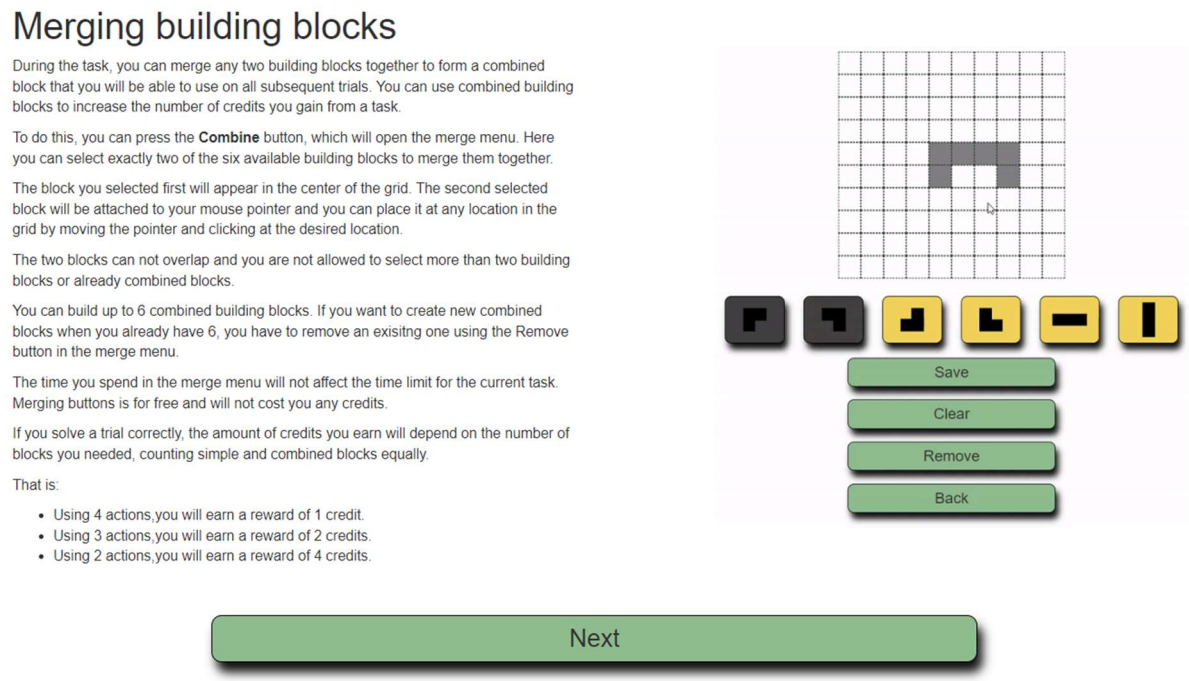
If the participant clicks the Enter button, the selection is evaluated and a the display flashes either in green or in red for a short time depending on whether the selection was correct or not (i.e. the correct four building blocks that are contained in the silhouette were clicked before hitting enter).

If the time limit is exceed, the trial is marked as failed and the next trial is loaded.

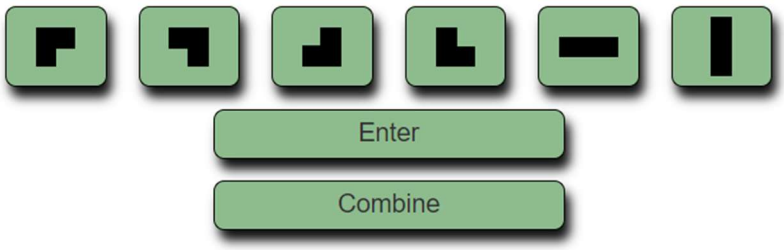
2.3 Split and Merge Option

There is an additional option in the main task that can be made available for the user by choosing the according option in the configuration page (See Chapter 3). If this option is available, the participant can combine two building blocks and form a new button that can be used to select the two primitive building blocks at once.

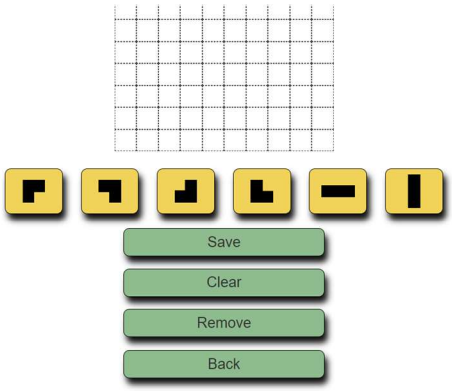
If the split merge option is active, an additional introduction page is shown before the main task starts:

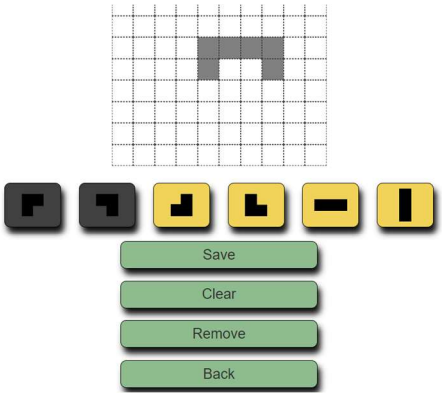


In the main task, an additional “combine” Button is available.

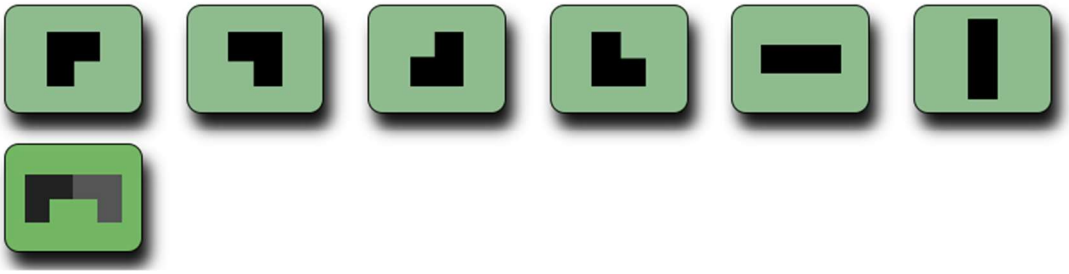


Pressing the combine button will open a new window, where the participant can click on two building blocks. The first block will be displayed in the center of the grid, the second block can be placed anywhere in the grid by clicking on the according cell.



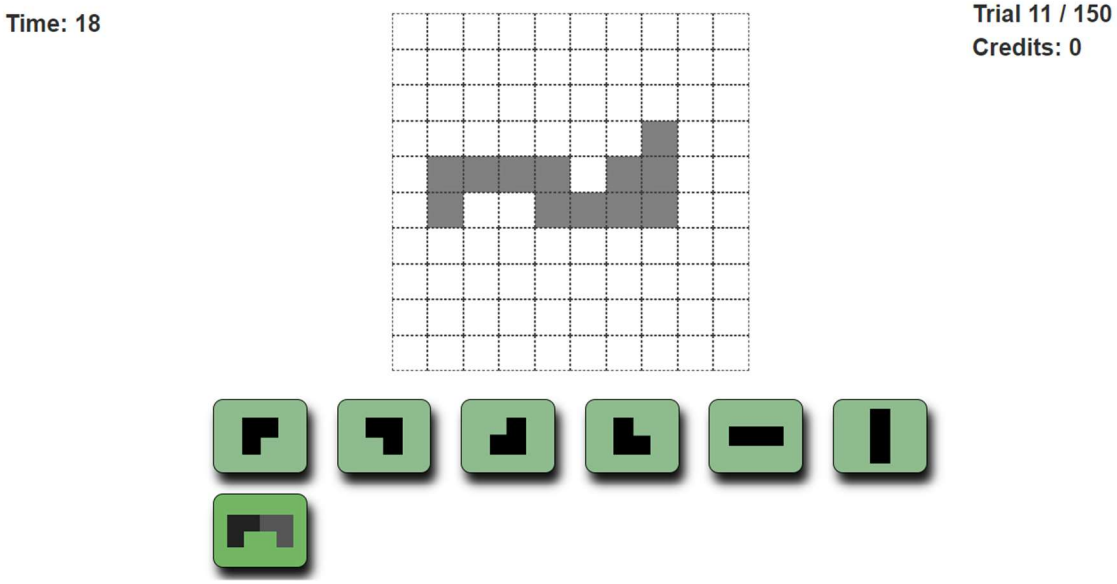


After pressing save, the new chunk will be available in addition to the buttons in the main task:



Participant can build up to six new combined building blocks. There is no time limit while in the split merge menu and the timer for the current main task trial will be paused. The combined button can be used in the same way as the other buttons but will count as two selections. The selection is only correct, if both primitive blocks appear in the silhouette in the exact same relative position.

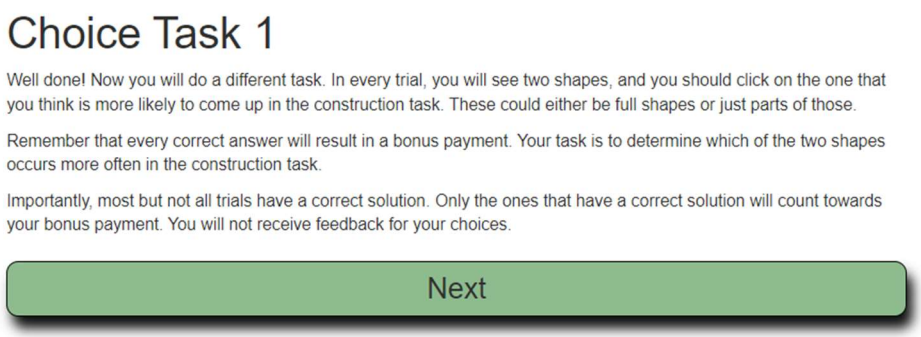
When using the split merge function, the participants can earn credits. The current credit score is displayed in the top right-hand corner. As a motivation to use the split-merge feature the number of credits that can be earned in one trial depends on how many buttons the participate used for selecting the blocks. A task can be solved by using four, three or only two buttons and the less button clicks are needed, the more credits can be earned.



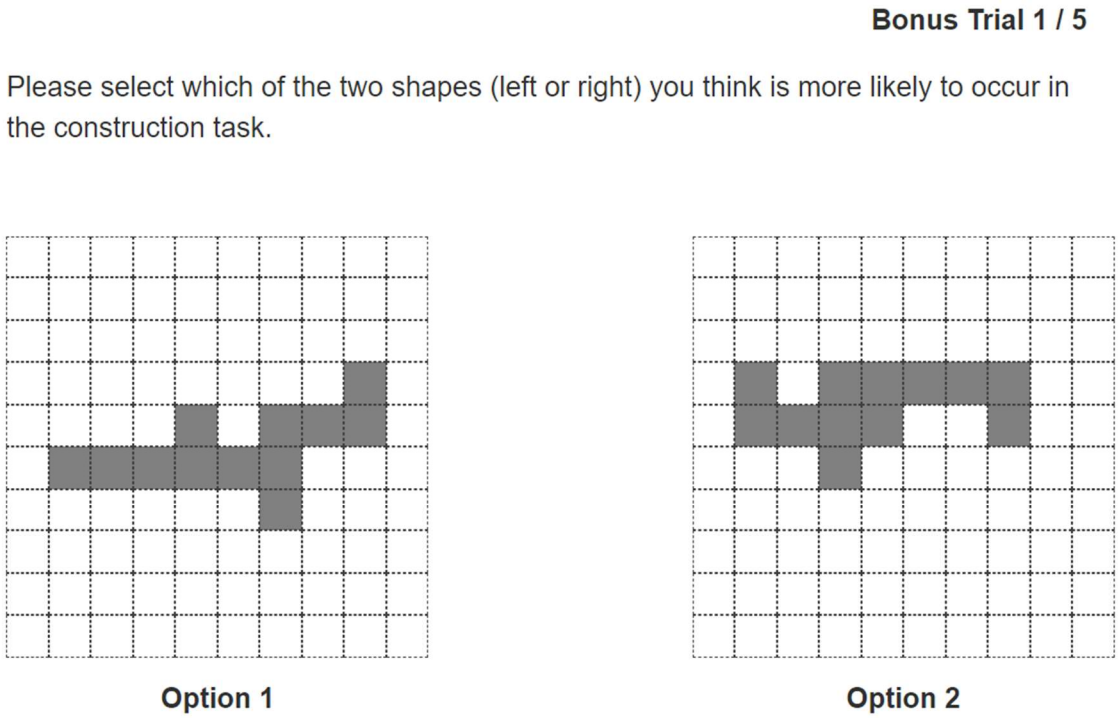
Is the selection wrong or the time limit reached, the participate will not gain any credits for this trial.

2.4 Debriefing / Choice Task

Instruction page for this task:



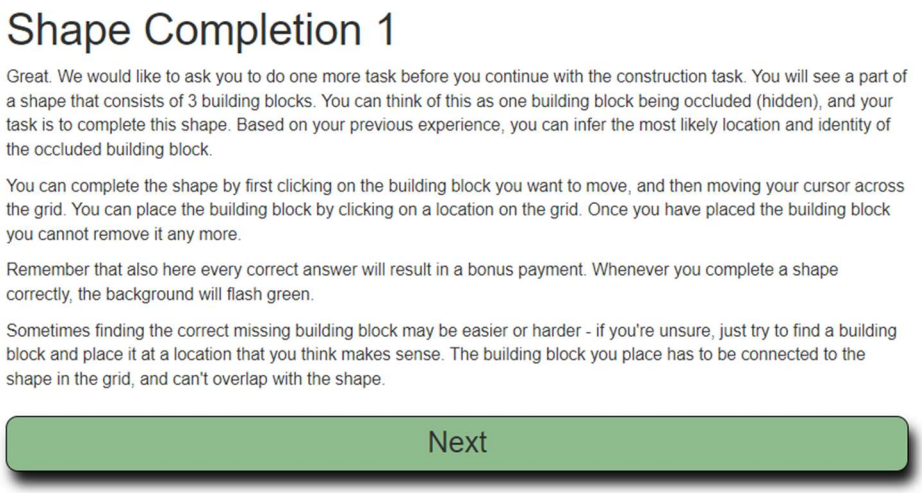
In this task the participant is shown two different silhouettes (Option 1 and Option 2). By clicking in one of the images the participant has to decide which of the shapes is more likely to occur in the main task.



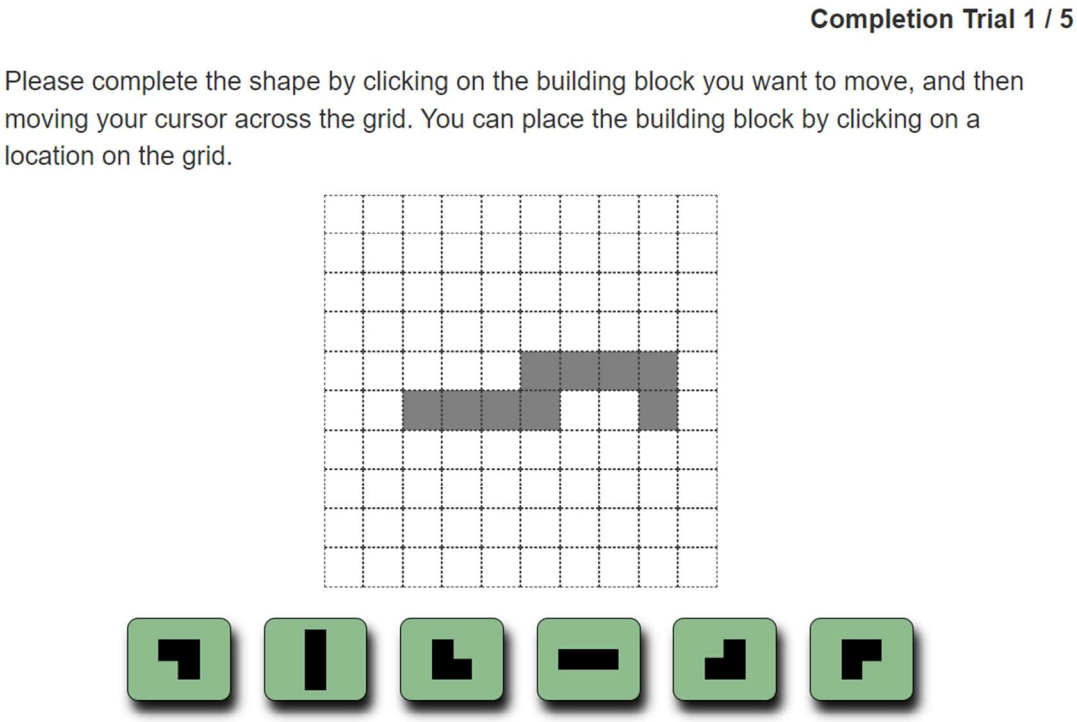
There is no time limit and also no feedback about whether the answer was wrong or right. In the top right-hand corner the current trial and the number of total trials of this tasks is displayed.

2.5 Shape Completion / Minesweeper Task

Instruction page for this task:



In this task again a silhouette is shown to the participant. This silhouette only contains of three instead of four building block. The participant has to “complete” the silhouette by placing one of the six building blocks at a certain location in the grid.



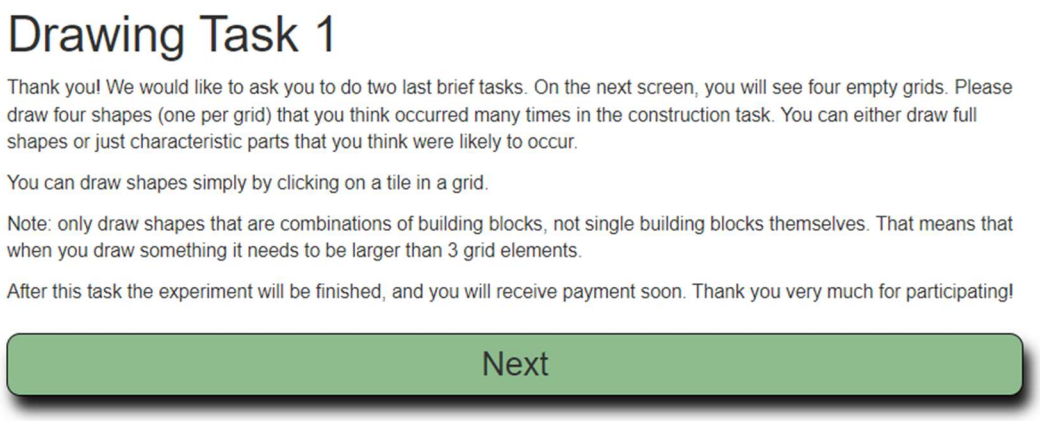
When clicking at a building block, the shape of this building block is attached to the cursor. By hovering over the cells in the grid, the participant can choose a location for the block. By clicking a the currently selected cell, the building block is placed and the answers is committed.

If the correct building block was placed to the correct position a positive feedback is shown by flashing the display green. Negative feedback on wrong selections is not provided.

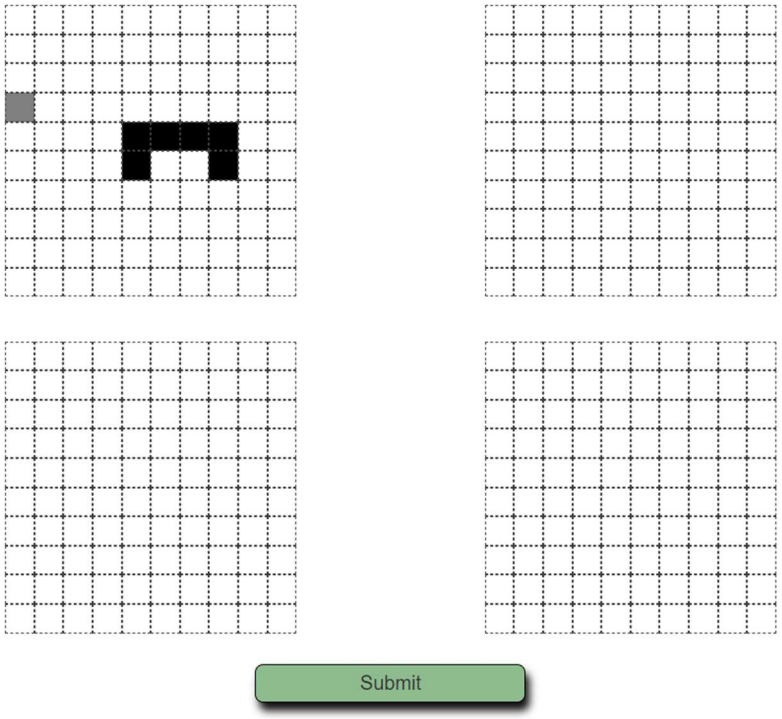
Again ,there is no time limit to this task. The current trial and the total number of trials for this task is shown in the top right-hand corner.

2.6 Drawing Task

Instruction page for this task:



In this task, four empty grids are displayed. The participant is told to draw certain shapes that are likely to occur in the main task.

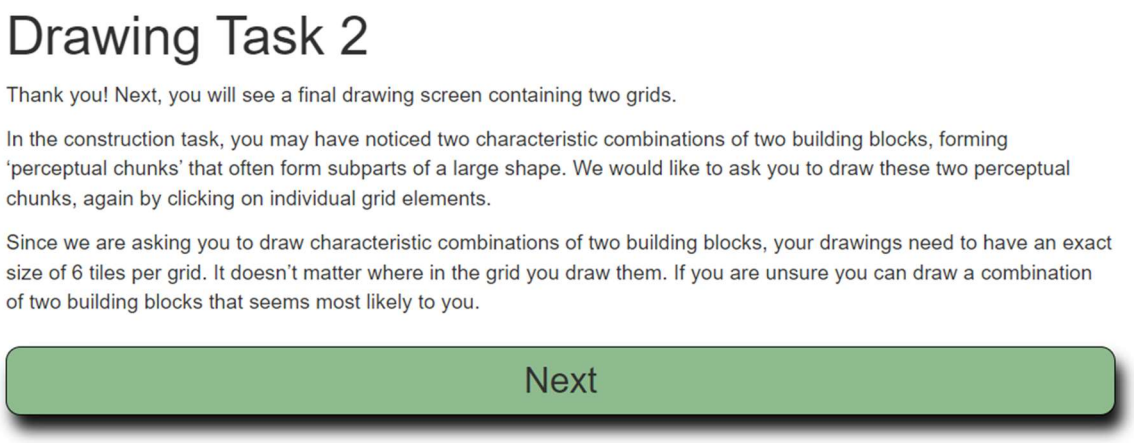


By hovering over a cell in the grid with the cursor, the selected cell is highlighted in grey. By clicking at this grid cell, the according cell is filled out with black color. When the participant finished the drawings they are committed by pressing on the submit task. The submit button is active all the time, so the participant can also decide to draw less than 4 shapes or no shape at all.

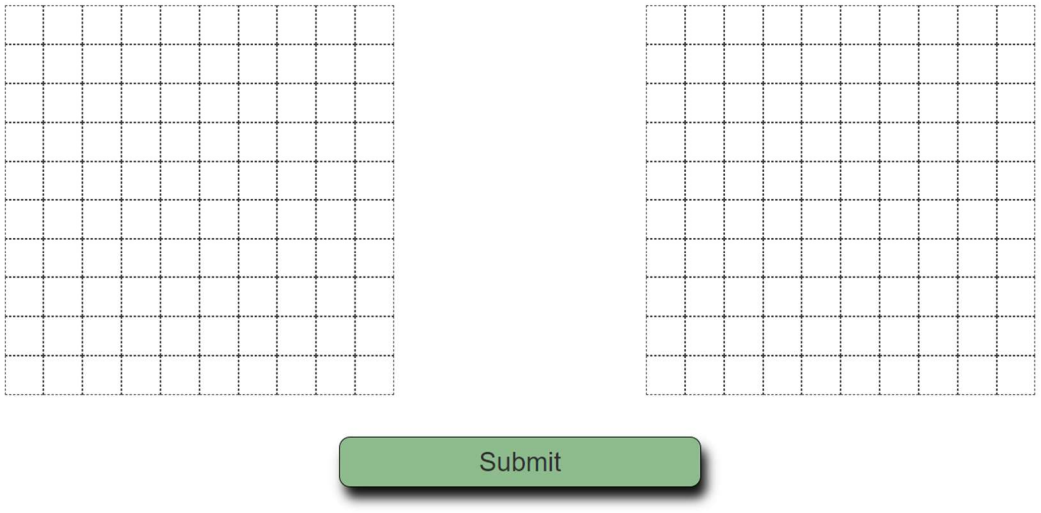
There is no time limit for this task and there are no multiple trials.

There is a second part of the drawing task following right after the first task

Instruction page for this task:



. Here, two empty grids are displayed. This the instead of drawing any kind of shape the user is explicitly asked to draw a combination of two building blocks in each grid (So in each grid the participant should fill in exactly 6 cells).



The drawing works exactly the same as for the first drawing task. Again, there is no time limit and no repeating trials.

2.7 Comment Section

In the very last part of the experiment, the participant can leave a comment or feedback about the experiment:

There is no time limit and no limitations on the length of the comment. The participant can also continue without writing any comment at all. Clicking on the button will end the experiment and the participate will be redirected back to prolific. The url to which the participant contains a certain parameter, proving that the experiment was finished successfully. This is important for doing the payment later.

3 Configuration

The general procedure and other aspects of the experiment can be customized via a configuration page. The page is available under:

http://[URL_TO_THE_EXPERIMENT_PAGE]/config.html

depending on where the website is deployed (or if it run locally), fill in the according route to the experiment page.

It is possible to run multiple experiments with different configurations at the time. To do this, it is possible to create, load and modify separate configuration files independently from each other and later choose which of them should be used in a certain session by setting a parameter in the experiment URL accordingly.

Ther first section of the configuration page provides functions to manage different configuration files.

To load an existing configuration, select it in the drop-down menu and press “Load”. All the input-fields will be filled in with the current values of this configuration, and all changes will overwrite the values in the selected configuration file.

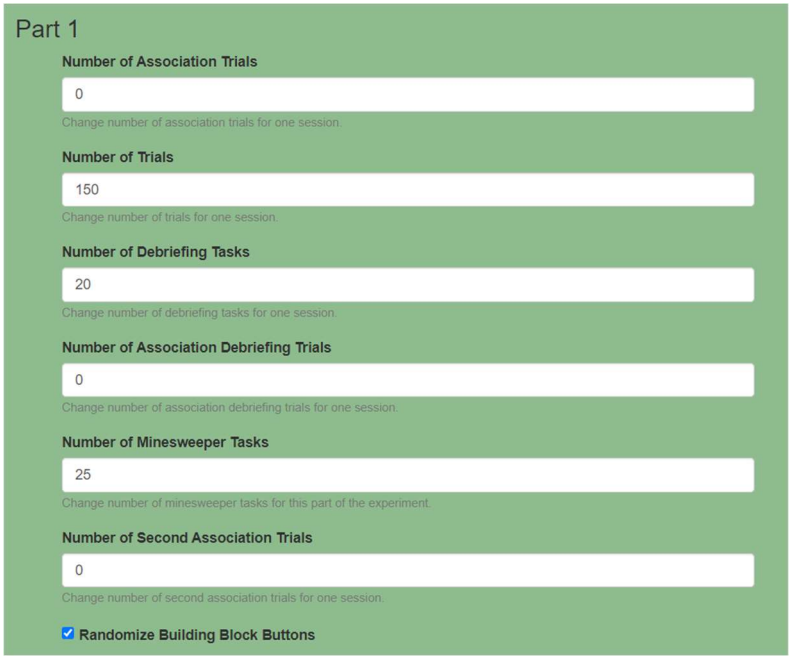
To create a new configuration file, choose a name which is not already in use and press “Create”. All input-fields will be filled with default values and you can set them based on the desired requirements for the new experiment.

In the second section, general settings for the experiment can be set:

For changing the background color of the website, type in any color as a HEX Code. The time limit for the main task can be set in seconds. The fields “Directory” and “Stimuli Set” are used to define which stimuli should be used in the experiment. For further information about how stimuli are loaded see Chapter 5.

By selecting or deselecting the two checkboxes in bottom, you can toggle whether the “Split and Merge function” and the Drawing Task should be used in the experiment. For details about those options see Chapter 2.3 and Chapter 2.6.

In the last section of the page, you can define which tasks should be used in the experiment and how many trials each task will have.



The block of input fields, shown in the screenshot above, appears three times on the configuration page. This is because all the tasks can be repeated in up to three experiment parts while every part can be configured differently. By typing a 0 in one of the input fields, this task will be skipped in the current part. Otherwise, the task will be used contain the number of trials that is specified in the field. If all tasks in a part have the entry 0 this part is skipped entirely. With that, it is possible to create arbitrary combinations of tasks which are repeating, occurring once, or are left out completely.

The checkbox at the end of each part defines whether the building block buttons in the main task should be shuffled or always are displayed in the same order.

Make sure to always check which configuration file you edit and that you have to load a config file before you are able to modify it. After pressing the “Submit” button at the very end of the page the selected configuration file is overwritten (or created in the case of a new one) with the current values of the input fields. The changes are active after the next reload of the experiment page.

To define which configuration should be used you can set a directly in the parameter in the URL:

http://[URL_TO_THE_EXPERIMENT_PAGE]/?CONFIG=[configname]

And replacing [configname] with the name of the config file that should be used for this session.

4 Code

4.1 Overview

Repository URL: https://gitlab.tuebingen.mpg.de/pschwartenbeck/tangram
URL to experiment page: http://134.76.24.67/
URL to config page: http://134.76.24.67/config.html
Database: mysql on port **3306** (Database: **experiment** User: **root** pw: **tangram**)

This code project is a web-application, running on an apache webserver.

For the frontend, the following programming languages and frameworks are used:

- **HTML**
- **CSS**
- **JavaScript**
- **Bootstrap**
- **JQuery**

For the backend, the following programming languages and frameworks are used:

- **NodeJS**
- **NestJS**
- **JavaScript**
- **TypeScript**
- **SQL**

For helper scripts, result evaluation and data management, the following programming languages and frameworks are used:

- **Python**

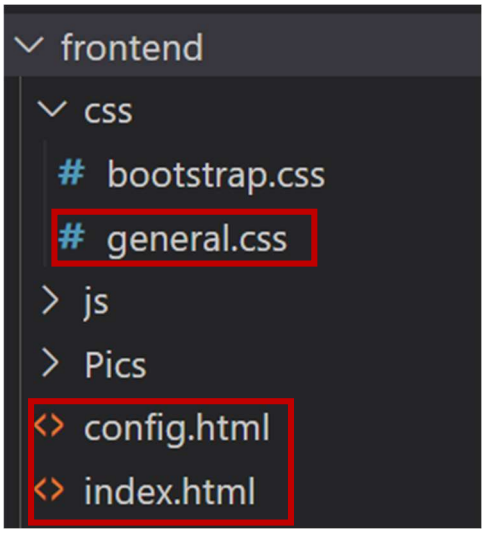
4.2 Frontend Code

Structure and Styling of the Website are contained in a single html and css file each.

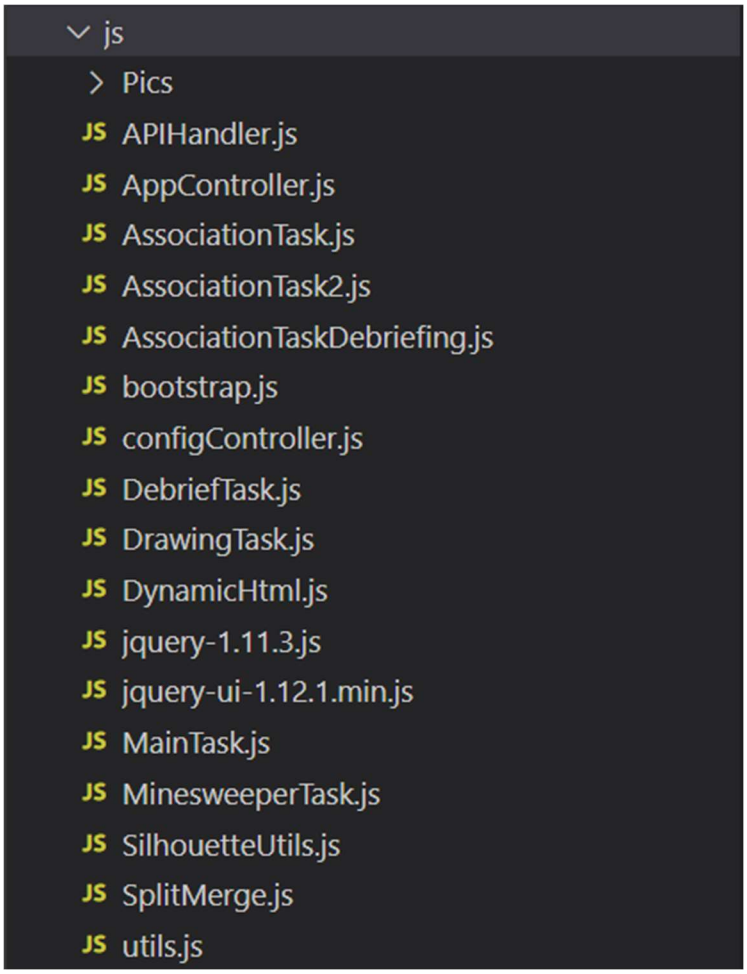
general.css
Contains all css styles used in the entire application

index.html
Contains the html structure of all UI components and pages that are shown during the experiment. Transitions between different tasks (see Chapter 2) are done by dynamically showing and hiding containers defined in the **index.html** file. Some UI Elements are not described in the index.html but are created dynamically using JavaScript (jQuery) code. Examples for this are the building block buttons, the grid and the silhouettes in the main task.

config.html
Contains all html structure for the configuration page (See Chapter 3).



The subfolder **js** contains multiple JavaScript files that handle the entire frontend behavior of the website (Handle user inputs like button clicks, transition between experiment parts, sending http requests to the backend to load and save data. In the following the most important Javascript files are listed with a short description of their functionality:



AppController.js

This file handles the main application flow including initialization, page transition and click listeners.

The **init()** function is called in the very beginning and handle all functionalities that have to run when the page is loaded. Here for example the configuration that should be used for the current session is loaded. This configuration affects the appearance of the website as well as the general procedure (which tasks are active, how many trials etc. See Chapter 3).

Some functions handle the **time limit** in the main task using a timer object that is started, paused and resumed depending on the task logic.

The function **next()** handles the main flow of the experiment and is responsible for transitioning between different tasks and different trials in a task. The function consists of a big switch-case block based on the **state** variable and execute certain code based on the current state (what is the current task) and the configuration (how many trials are left, what is the next task). The **next()** function is called after every trial in a task and either load the next trial (if the maximum number of trials is not reached), transition to the next task (if the maximum number of trials is reached), transition to the next experiment part (if the last task of the current part is over, See Chapter 3) or ending the experiment if the (last task is over).

The rest of the file contains assignments of click and hover listeners for different UI components. The listeners are either directly defined and enable or disable certain UI elements or another function is called. Those functions can be located in other JavaScript files.

```
function set_button_click_listeners(){

    // welcome --> consent
    $("#welcome__button").click(function(){
        $(".welcome").slideUp();
        $(".consent").slideDown();
    });

    // consent --> return-hit
    $("#consent__button__disagree").click(function(){
        $(".consent").slideUp();
        $(".return-hit").slideDown();
    });
```

```
// split merge
$("#combine_button").click( function() {
    on_combine_button_clicked(session)
});

$("#save_split_merge_button").click( function() {
    on_save_split_merge_button_clicked(session)
});
```

APIHandler.js

This file contains several methods handling the entire communication with the backend (API). The URL for all API calls is saved in a constant and used in every function. Based on if you run the application on the server or locally on your machine, you might change this URL accordingly.

```
const API_URL ="http://134.76.24.67/node" //"http://127.0.0.1:3000"
```

According to what data should be read or written, different routes are called. For example to get all sessions that are saved to the database, a GET request is sent to the URL: http://134.76.24.67/node/session

You can also call this URL in any browser and check the response there.

Function that load data from the server usually send a **GET** Request to the server and then parse the result in any way it is needed by the frontend application. When writing to the server, a JSON object is created, containing key-value pairs with all necessary fields which is then sent in a **POST** request.

```
        // send Post request to API
        xhr.open("POST", API_URL+"/trial");
        xhr.setRequestHeader("Content-Type", "application/json");
        xhr.send(JSON.stringify({
            session: session,
            silhouette: silhouette,
            timestamp: timestamp,
            reward: reward,
            experiment_part: experiment_part,
            trial_type : trial_type,
            silhouette_str: silhouette_str
        }));
    });
```

DynamicHtml.js

This file contains all methods to dynamically build or modify UI elements. The functions build html elements and append them somewhere in the **index.html** file or manipulate certain properties of existing elements using jQuery commands.

For examples all grids and the silhouette rendering in the grids is done by functions in this file.

```
/**
 * Build grid for the main task in html file
 * @param {number} grid_max
 */
function build_grid_main_task(grid_max){
    for(var i =0; i<grid_max; i++){
        for(var j=0; j<grid_max; j++){
            $('#grid').append(
                $('<div>').prop({
                    id: "grid_"+i+'_'+j,
                    innerHTML: '',
                    className: 'grid_cell'
                })
            );
        }
    }
}
```

configController.js

This file handles all functionalities for the config page. It is the only JavaScript file in the folder which is not connected to the **index.html** file but to the **config.html** file. The functions in this file are responsible for creating, loading and saving certain configuration files, reading out values of the input fields and setting values to the input fields.

utils.js and SilhouetteUtils.js

Those two files contain general functionalities that are used in different parts of the application. SilhouetteUtils especially contains methods to display building blocks and whole silhouettes in the grids.

MainTask.js, DrawingTask.js, …

All the other JavaScript files contain the code to handle the different tasks of the experiment. Every single task has its own JavaScript file that handles all the functionalities that is used by this file. Only transitions between tasks and trials are not handled by those files but are handled in the **AppController.js** file. Since all files are connected to the **index.html** file, the methods declared in one JavaScript file can be called from any other file, that was loaded after it. Whenever a new JavaScript file is created, it has to linked to the index.html file as well (make sure to load it before all the files that use functions from it).

See the loading order in the **index.html** file:

```
<!-- Custom files -->
<script type="text/javascript" src="js/utils.js"></script>
<script type="text/javascript" src="js/SilhouetteUtils.js"></script>
<script type="text/javascript" src="js/APIHandler.js"></script>
<script type="text/javascript" src="js/DynamicHtml.js"></script>
<script type="text/javascript" src="js/SplitMerge.js"></script>
<script type="text/javascript" src="js/MainTask.js"></script>
<script type="text/javascript" src="js/DebriefTask.js"></script>
<script type="text/javascript" src="js/MinesweeperTask.js"></script>
<script type="text/javascript" src="js/DrawingTask.js"></script>
<script type="text/javascript" src="js/AssociationTask.js"></script>
<script type="text/javascript" src="js/AssociationTaskDebriefing.js"></script
<script type="text/javascript" src="js/AssociationTask2.js"></script>
<script type="text/javascript" src="js/AppController.js"></script>
```

4.3 Backend Code

The backend, i.e. the application that runs on the server, is build in NodeJs. On top, the framework NestJs is used which is a NodeJS framework for developing lightweighted server applications to handle API calls.

To start the backend application locally on your computer (or any server), **install NodeJs**, then navigate to **tangram/backend** with any command shell and run

npm install

to install all necessary packages / libraries. After the installation is done, you can start the application by executing the command

npm start

When running the app on the server you have to make sure, the process is not stopped, when you close the console or log out, therefore, for running the app on the server, navigate to **tangram/backend** and run the following command.

forever start dist/main.js

You may have to install the forever package first. To stop the running processing you can use:

forever stop (index)

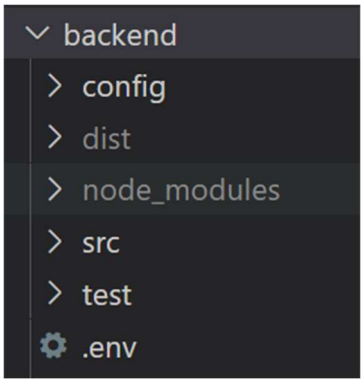
The index of the application can be found out by running

forever list

but as long as you do not have started other apps using forever, it should always be 0.

To properly run the backend application, you also need a mysql database contain all necessary tables. For further information see Chapter 5. By default the application will listen to the **port 3000** and this is also the case for the currently running experiment backend on the server. The webserver is configurated such that all http requests incoming at the route http://134.76.24.67/**node** will be redirected to **localhost:3000** and therefore handled by the running NodeJs application. For further information see Chapter 6.

The backend folder is structured as follows:

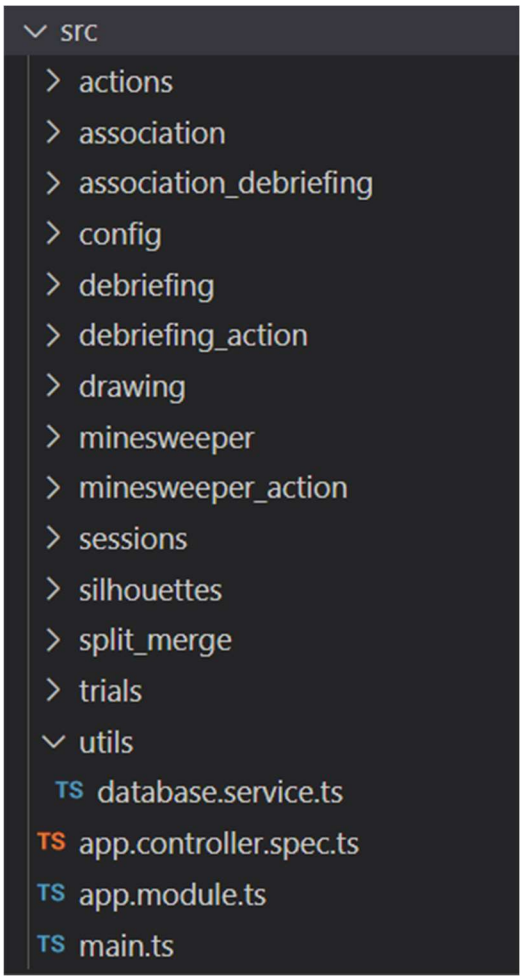


The **config** folder contains all configuration json files that were created via the config page. When adding a new config or updating an old one, a new json file is created or an existing one is overwritten.

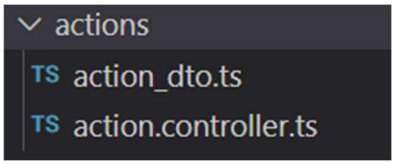
The **src** folder contains all the code handling the incoming HTTP request and the connection to the database.

The **.env** file contains login credentials and other information about the database connection. Change those entries accordingly when deploying the application locally or on another server.

Lets have a look at the structure of the actual backend code contained in the **src** folder:



As you can see, the code to handle requests for different functionalities or tasks of the website is divided into subfolders. Every subfolder contains one **controller** file which listen for requests on a certain route and one or multiple **dto** (data transfer object) files which describe the structure of all objects that are received or sent via http.



To learn more about the structure of a NestJS application, check their documentation.

The **dto** files are simple typescript classes that just contain all attributes of a certain object. For example, whenever a user select a button in the main task, an **action object** is sent to the backend which have the following structure:

```
export class ActionDTO {
    trial: number;
    building_block: String;
    reaction_time: number;
    selected: boolean;
}
```

Just make sure that every attribute has the correct datatype and the **same name** as the attribute in the json file that is sent from the frontend application (This is important to map the attributes from the json object to the typescript object).

A controller file usually looks like this:

```
@Controller("action")
export class ActionController {
  constructor(private readonly dbService: DatabaseService) {}

  @Get()
  async getAllActions(){
    return await this.dbService.getAllActions();
  }


  @Post()
  saveRecord(@Body() action: ActionDTO){
    // receives action as JSON object and save it to the database
    this.dbService.saveAction(action);
  }
}
```

The annotation above the class definition sets the route on which this controller should listen. For example the controller in the screenshot above handles all incoming HTTP requests to the route

http://134.76.24.67/node/**action**

and only them. Inside the class we define a function for every request type we want to cover which is marked by the annotations above the function heads (In this case @Get and @Post). In this example the controller will only accept GET and POST requests from the frontend.

Inside those function we can define the behavior of the backend when a certain API call was received.

Almost all files in the **src** folder have the exact same structure and all the controllers are working in the same way, only handling the requests to a specific route (most of the times one route corresponds to one task of the experiment)

The only exception is the file **database.service.ts** located under **src/utils**. This file is responsible for reading from and writing to the database. The function connectToDatabase tries to create a connection to a mysql database using all the settings specified in the **.env** file.

```
async connectToDatabase(){

  // setup a connection to the mysql database and save it as global variable
  var mysql = require('mysql');
  this.con = mysql.createConnection({
    host: this.configService.get<string>('DB_HOST'),
    port: this.configService.get<string>('DB_PORT'),
    user: this.configService.get<string>('DB_USER'),
    password: this.configService.get<string>('DB_PASSWORD'),
    database: this.configService.get<string>('DB_DATABASE')
  });
  console.log("connecting to database...")
  await this.con.connect();
  console.log("connected");
}
```

The function is called as soon as the application starts and uses the same database connection until the application is shut down or the database is no available anymore.

All other functions in the file handle a certain database action like reading from a certain table or creating a new entry. They all look something like this:

```
async getAllActions(){
  if (!this.con){
    await this.connectToDatabase();
  }
  var sql = `SELECT * FROM action`
  var actions = await this.selectQuery(sql);
  return actions;
}
```

If there is currently no database connection, the connectToDatabase() method is called. If the connection is established a certain SQL command is executed and the result is returned.

There is basically not more functionality to the backend than what we saw so far. In all cases a certain **HTTP request** is sent to a specific route and handled by the according **controller**. **POST** requests usually contain a **json object** as payload which is parsed to a **typescript object** using a **dto class**. In all cases the controller then calls a certain function of the **database.service** to either **read** from the database, **create** a new record or **modify** an existing one.

The **main.ts** file is the called to start the application. Here you can change the **port** (**currently 3000**) on which the application will accept incoming requests.

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors();
  await app.listen(3000);
}
bootstrap();
```

In the **app.module.ts** file every controller or service that is used in the application must be registered:

```
@Module({
  imports: [ConfigModule.forRoot({isGlobal: true})],
  controllers: [ActionController, TrialController, ConfigController, SilhouetteController, SessionController,
  providers: [DatabaseService],
})
export class AppModule {}
```

Whenever you create a new controller or service, you have to **make sure to add the class to the according list in this file**! Otherwise, a controller will not listen to the route you specified and a service will not be available to any controllers.

Please Note:

Whenever you make changes to the backend code, you have to restart the application to make the modified code available!

5 Database and Loading Stimuli

To save the experiment data we are using a mysql database. In order to run the backend application without errors and to properly save and load all data that is produced during the experiment, you have to make sure that:

1) The mysql server is running
2) The backend application can connect to the database (for this see Chapter 4.3 and Chapter 6)
3) There exist a database contain the correct tables with the correct structure.

To ensure point 3) there is a **dump file** in the code repository called **db_dump_tables.sql**. The file does not contain any data but the sql code to create a database and all the necessary tables.

Currently the following tables are used by the application:

Most of the tables and their columns are self-explanatory but let's look at some of the tables that are important to understand:

Session

Every time the website is loaded, a new session is created. The session will stay the same until the experiment is over or the website is closed. Therefore it uniquely identifies one participant doing one experiment.

1	**session_id**	**INT**
2	participant_id	TEXT
3	code_version	INT
4	exit_quiz	MEDIUMTEXT
5	comment	MEDIUMTEXT
6	split_merge	INT
7	credits	INT

Every session has an incrementing **session id** as its primary key. Additionally a **participant id** is saved. This id depends on how the participant was redirected to the website. Until now, the experiment was only performed via **prolific**. When a participant is redirected to the experiment URL a parameter is appended to the URL containing a unique participant id. This id is saved to the database and is very important to identify the participant later and manage payment. When the experiment is over, the participant is redirected back to prolific. This redirection also have to contain the participant id to give the information back to prolific, that the participant finished the experiment successfully.

The columns **code_version** and **exit_quiz** are currently not used.

If a participant decides to leave a comment after the experiment it is saved as a string to **comment** column.

When using the split-merge function (see chapter 2.3) a True-flag is saved to the **split_merge** column and in this case, also the amount of **credits** a participant gained is saved.

To handle the main task, several tables are required:

Trial

The main task consists of multiple trial and in every trial a different silhouette is shown.

1	**trial_id**	**INT**
2	session_id	INT
3	silhouette_id	TEXT
4	time_stamp	BIGINT
5	reward	FLOAT
6	timeout	INT
7	experiment_part	INT
8	trial_type	INT
9	silhouette	TEXT

For every trial a record is saved, containing the session_id and the id of the displayed silhouette. Also the time the participant needed to do the trial and the reward (1 for success 0 for fail) is saved. If the trial ended in a timeout this is saved as a True-flag. Because the main task can be split into different parts, the current part id is also saved. Every trial/silhouette has a certain trial type which is saved as well as the silhouette itself in a stringyfied version (more on this later).

action

Whenever the participant clicks on a building block button, an action is saved to the database.

1	**action_id**	**INT**
2	trial	INT
3	building_block	TEXT
4	reaction_time	DOUBLE
5	selected	INT

It contains the current trial id and the id of the building block that was clicked. The reaction time shows the time in seconds that has passed since the last click or in case of the first click since the start of the trial. Because the participant can deselect a selected building_block by clicking the button again, it is saved as a Boolean flag whether the block was selected or deselected.

Silhouette

This table contains the silhouettes that are displayed in the grid for every trial.

#	Name	Datentyp
1	silhouette_id	INT
2	solution	TEXT
3	trial_type	INT
4	stimset	TEXT

One silhouette consists of multiple building blocks, where each block has an id and a location on the grid. A record in this table looks like this:

silhouette_id	solution	trial_type	stimset
0	[{"x":3, "y":4, "id":0},{"x":5, "y":2, "id":1},{"x":2, "y":5, "id":2},{"x":5, "y":4, "id":4}]	0	HierarchiesStimSet2

The column solution is a stringyfied list of json object. Each object is one building block and has the three properties id, x and y. This string contains all the necessary information to render a silhouette to the grid. Beside that every silhouette has type which is also saved here.

It is possible to use multiple sets of silhouettes and change between them by setting the name of the stimuli set in the config (see Chapter 3). The Table contains all silhouettes from all stimsets, but according to what is set in the config, only the silhouettes from one specific stimset are loaded and used for the current experiment.

The silhouettes are not directly saved to the databased but are generated as csv-files containing one silhouette per row. This is also the case for other data for example the stimuli for the debriefing task or the minesweeper task. To save all those data to the database there is a python script called **update_databases.py** in the root directory of the repository. So whenever there are new stimuli that should be used in the experiment, the according csv files have to be saved in the repository under tangram/Stims to the according place in the folder structure. Then connect to the server, pull the repository and run the python script with

Python update_databases.py

This will automatically overwrite all database tables with the current stimuli in the Stims directory.

6 Server infrastructure

In the following a tutorial is provided how to install, configure and use the Apache webserver as it was done for this project. The tutorial will has a slightly different style than the rest of this documentation since it was copied over from a separate tutorial file which was created earlier.

Install node js

https://www.geeksforgeeks.org/installation-of-node-js-on-linux/

```
sudo apt install nodejs
```

```
sudo apt install npm
```

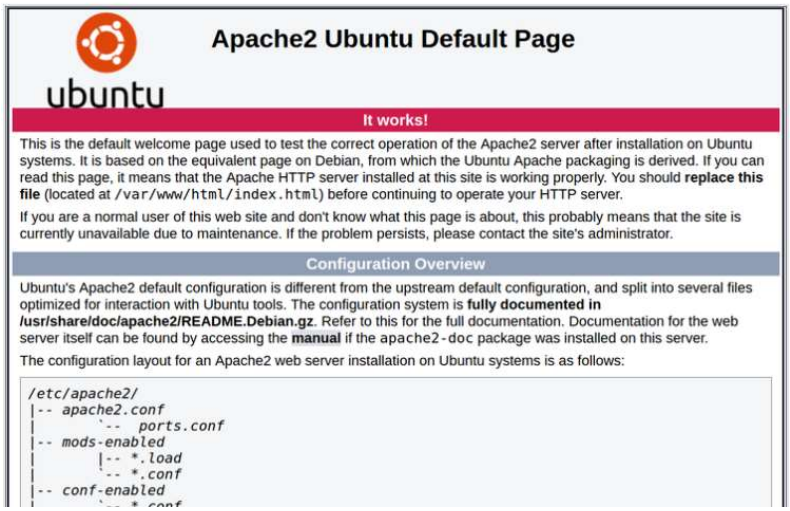
Install and configure Apache

```
$ sudo apt update
```

```
$ sudo apt install apache2
```

```
$ sudo ufw allow 'Apache'
```

Under http://your_ip you should get the following:



Config for the webserver:

Under /etc/apache2/sites-available/000-default.conf

- **Set Document root to the directory where the frontend application lies. Fore example:**

```
DocumentRoot /var/www/html/tangram/frontend
```

- **Add the following lines to ensure that your java script files are loaded correctly.** For example if your java script code is under /var/www/html/tangram/frontend/js then add the following lines:

```
Alias /javascript "/var/www/html/tangram/frontend/js/"
```

```
<Directory "/var/www/html/tangram/frontend/js">
        Options +Indexes
        AllowOverride None
        Order allow,deny
        Allow from all
</Directory>
```

  If the java script is still not loaded correctly, write the same lines to **/etc/apache2/conf-enabled/javascript-common.conf**

- Create a new config file, for example CUSTOM.conf under /etc/apache2/conf-enabled and add the following line:

```
ProxyPass /node http://localhost:3000
```

  Now all http requests with the prefix /node are redirected to port 3000.
  This is where your node js application is running. If you started the node js application with another port, just change 3000 accordingly. You can also you a different prefix than /node for example “/backend” or something else..

Install nest js

Use npm to install nestjs and use the “nest new” command to create a new nest js application:

```
$ npm i -g @nestjs/cli
$ nest new project-name
```

Start your application with **npm start**

Your server is now available under the port you set in the main.js (3000 by default)

See documentation under https://docs.nestjs.com/

For local development, if you get a CORS error (Cross origin resource sharing) make sure you add the following line in your **main.js**

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors();
  await app.listen(3000);
}
bootstrap();
```

Install MySQL

```
$ sudo apt update
```

```
$ sudo apt install mysql-server
```

```
$ sudo mysql_secure_installation
```

Create a user and set credentials by following this tutorial:

https://www.digitalocean.com/community/tutorials/how-to-install-mysql-on-ubuntu-20-04-de

Connecting Database to NodeJS Backend:

If you see the following error when trying to access the database in your node js application:

"Client does not support authentication protocol requested by server; consider upgrading MySQL client"

Solution:

https://stackoverflow.com/questions/50093144/mysql-8-0-client-does-not-support-authentication-protocol-requested-by-server

Execute the following query in MYSQL Workbench

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'password';
```

Where `root` as your user `localhost` as your URL and `password` as your password

Then run this query to refresh privileges:

```
flush privileges;
```

Try connecting using node after you do so.

If that doesn't work, try it without `@'localhost'` part.

Some Advice

- Install git on the server, develop the application locally and then pull your project on the server via git. Then set the document root of the Apache server to the directory where you pulled your application (or in the directory where your frontend code / index.html is located). If you have to change something, change the code locally, push and pull on the server and your changes are online immediately

- If some changes to the frontend code are not visible, clear your browser cash or reload the website with SHIFT + F5 to reload without cache

- If some changes to the backend code are not working but you don't get any errors, try to restart your nodejs application

- If you want your node js application to run in the background, even if you are not connected to the server anymore, try the forevery package: https://www.npmjs.com/package/forever
    - start (you have to be in the directory of your node js app): **forever start dist/main.js**
    - Get all running instances: **forever list**
    - Stop a process: **forever stop 0** (Replace 0 by the ID of the forever job you want to stop)