Introduction to z-Tree

Design: Urs Fischbacher

Programming: Urs Fischbacher & Stefan Schmid

This slides by Ernesto Reuben

How well do you know zTree?

1. **Never used it**
2. **Played with it a bit**
3. **Used it to run experiments**
4. **Used it to program and run experiments**

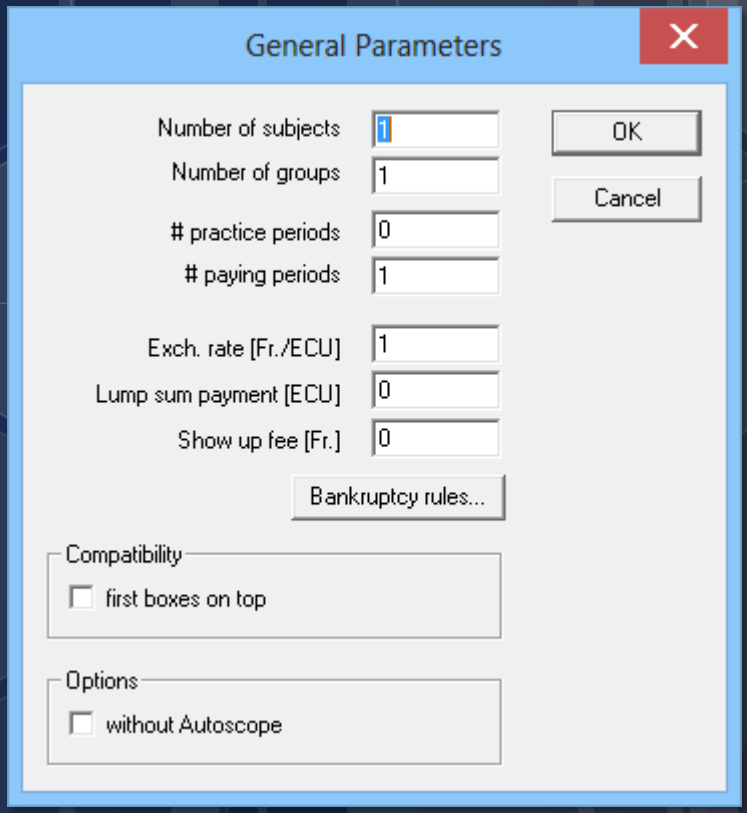
zTree online support

- zTree hompage:
  - http://www.iew.uzh.ch/ztree/index.php
- zTree Wiki
  - https://www.uzh.ch/iew/ztree/ssl-dir/wiki/
- zTree mailing list
  - go to https://lists.uzh.ch/iew.lists.uzh.ch/sympa/info/ztree and click on the 'Subscribe' link

zTree components

Treatment

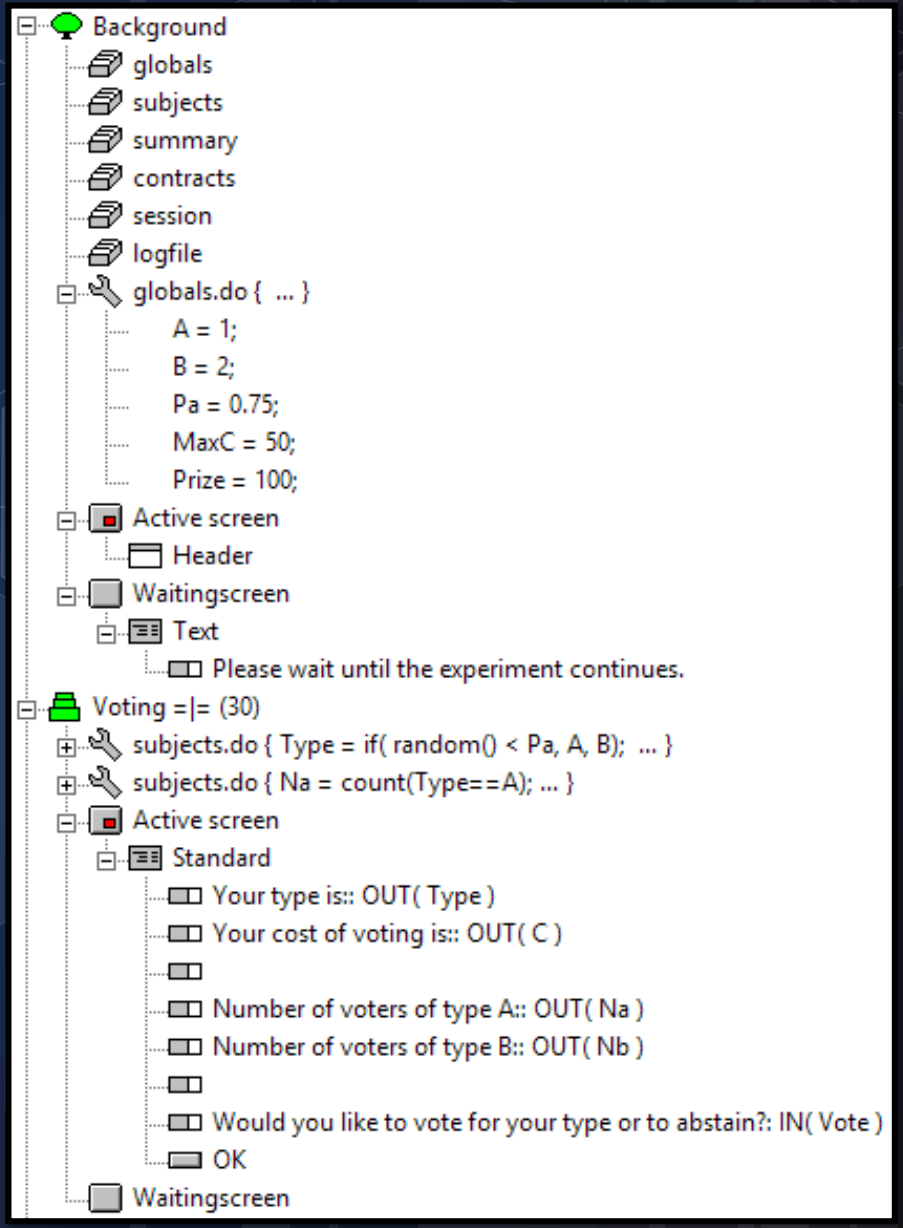
- Arranged in a tree structure (the stage tree)
- One **background stage**
  - Set number of subjects, groups, periods, exchange rate
  - Default screens
  - Treatment variables
- Any number of normal **stages**
  - Each stage corresponds (roughly) to one screen



zTree components

Stage

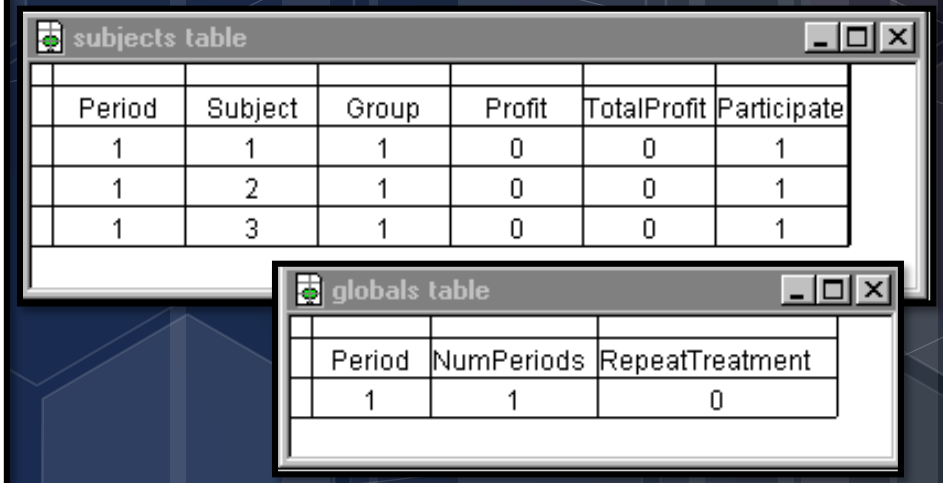
- **Properties**
  - When does it start and end
- **Programs**
  - Set and change variables
- Two **screens**
  - Active: for input and display
  - Waiting: display only
    - Screens contain **boxes** that in turn contain
      - **items** and **buttons**



zTree components

Tables

- Data are stored in **tables**. Mostly in
  - **subjects table**
    - One row per subject
  - **globals table**
    - One row per treatment (i.e., same value for all subjects)
  - A new subjects table and a new globals table are created in every period
- **Other tables**
  - summary, contracts, session, and logfile
  - Other tables can be accessed with the table.tablefunction



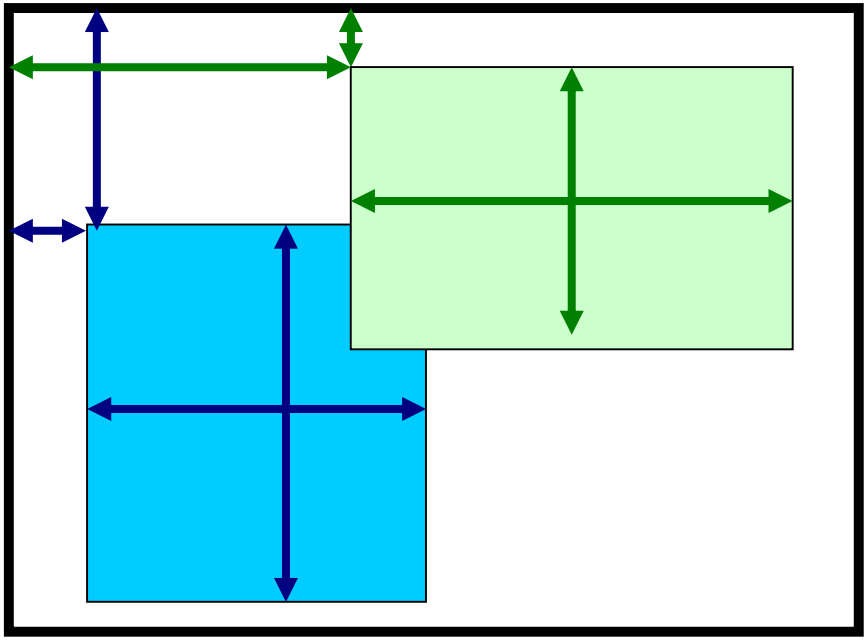
Programs

- **Programs** are executed at the beginning of a stage or when buttons are clicked
  - Calculations are done by z-Tree and then sent to the z-Leafs
  - Programs are executed **row by row** in the table they are called (i.e. subject by subject in the subjects table)

zTree components

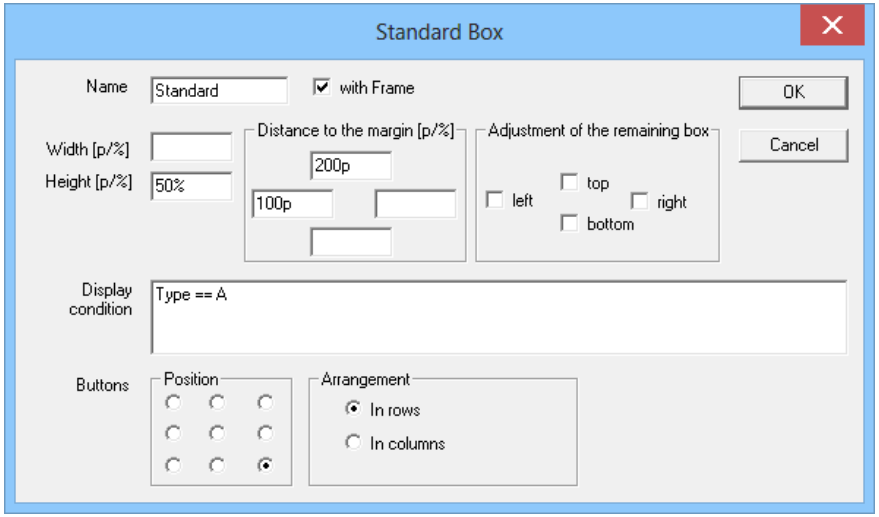
Boxes

- **Box** = rectangular area of the screen containing stuff
- They are positioned over each other
  - standard box, header box, help box, grid box, history box, chat box, plot box



Positioning boxes

- Distances can be set as % of the screen or in pixels



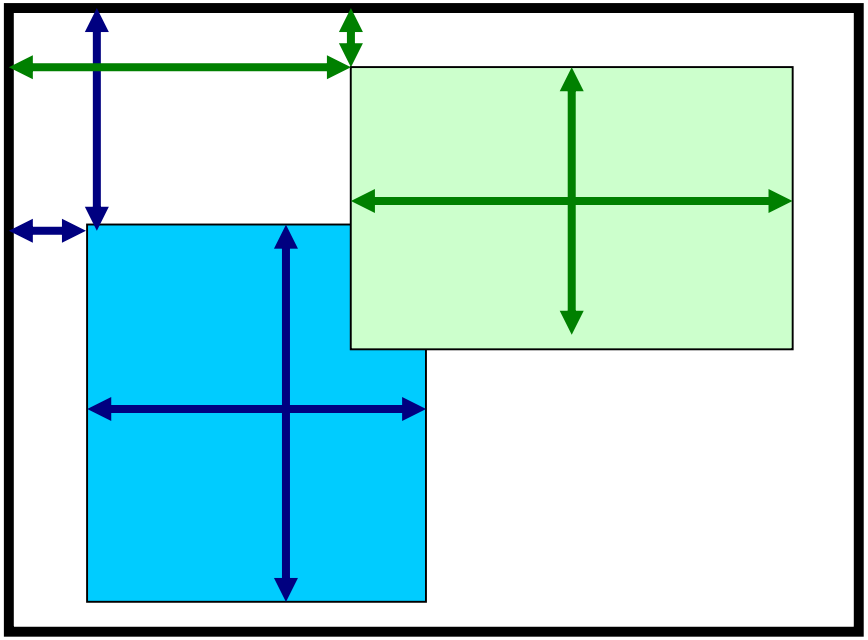
Display condition

- Makes boxes appear (when true) or disappear (when false)

zTree components

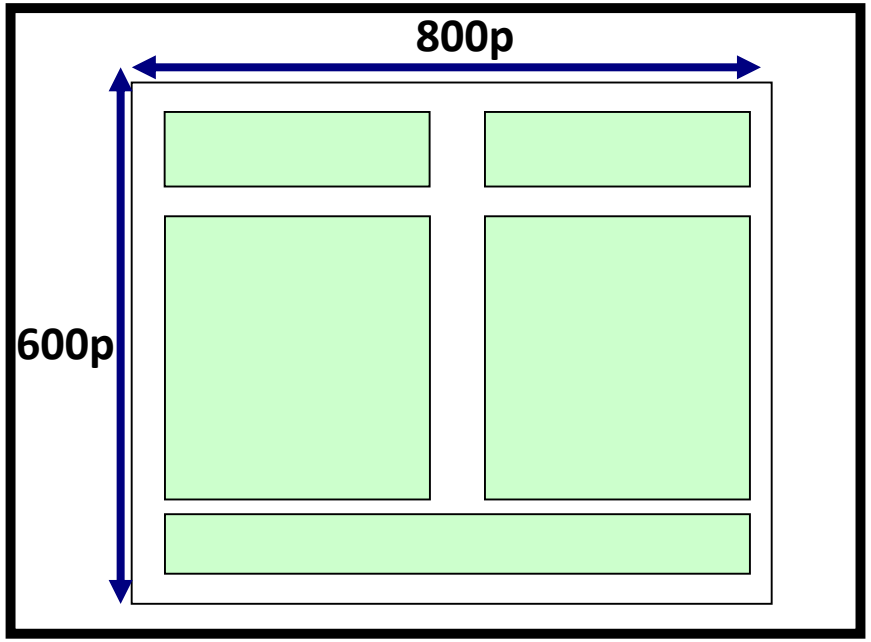
Boxes

- **Box** = rectangular area of the screen containing stuff
- They are positioned over each other
  - standard box, header box, help box, grid box, history box, chat box, plot box

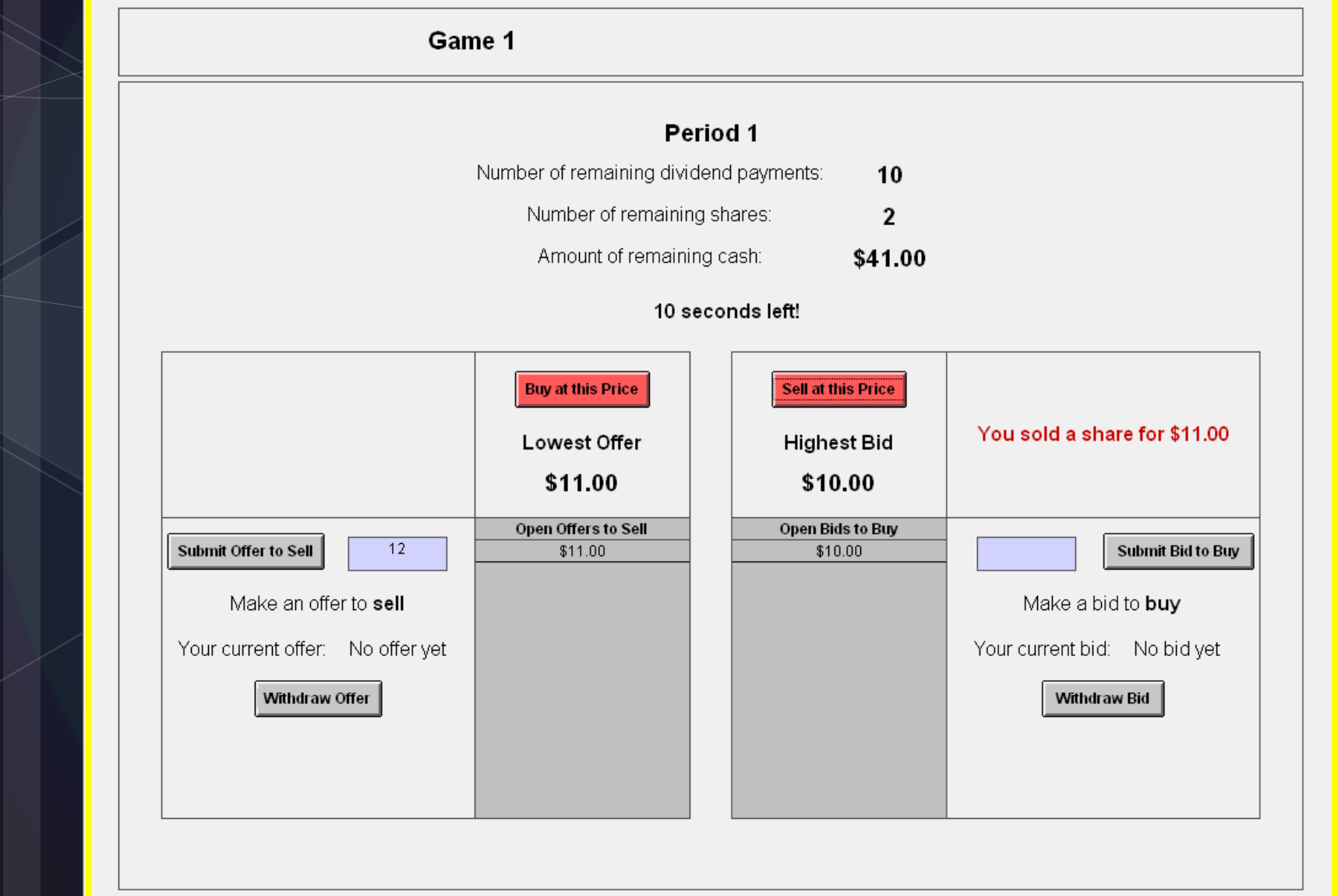


Useful tip

- Use **container boxes**
  - rectangular area containing other boxes
  - lets you move many boxes at the same time and keep things in place with different resolutions



zTree components



How to build a test environment

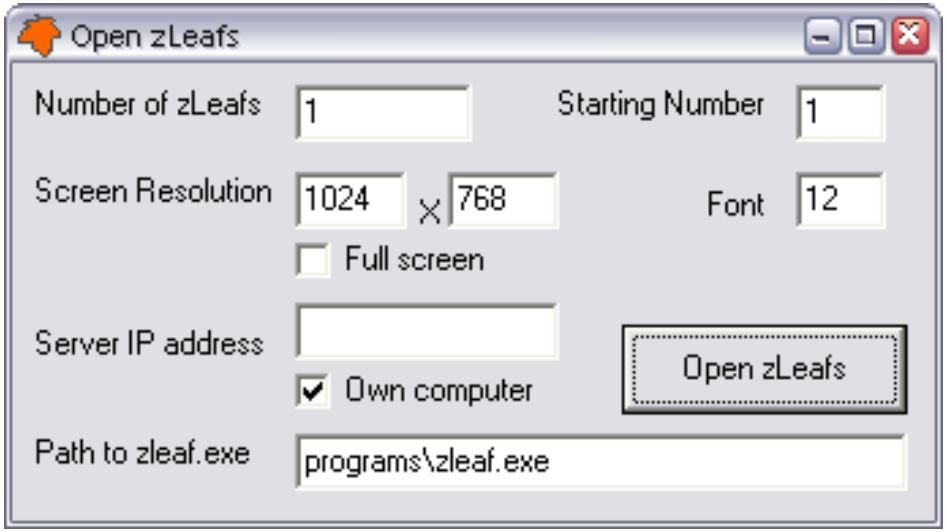
Unzip zTreeMaterials.zip into a folder.

- Can be found at http://ereuben.net/teach/zTreeMaterials.zip

Open zTree with the batch file: “openztree.bat”

Open the file: “Open Zleafs.exe”

- Set as many zLeafs as necessary
- If needed, change screen resolution and other options



Exp. 1: Rational turnout

Voters are randomly selected to prefer A or B

- Probability of preferring a $p_A > ½$

Voters can vote for A, B, or abstain

- They get 100 if their preference wins and 0 otherwise
- Voting is costly: costs are drawn from a uniform distribution $c_i \in [0, 50]$

To have a function al program we need:

- Set variables in the background stage
- Two other stages
  - Voting stage: voters are told their preference and make their decision
  - Result stage: voters are informed of the election's outcome

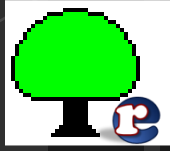
Creating variables

Variables are defined the first time they are referenced in a table

- They are always a real number

Defining treatment variables in the background stage:

```
globals.do{
    A = 1;
    B = 2;
    Pa = 0.75;
    MaxC = 50;
    Prize = 100;
}
```



Functions

There is a good number of functions that can be used for programming:

Draw types and costs:

```
subjects.do{
    Type = if( random() < Pa, A, B);
    C = round( random() * MaxC, 1);
}
```

Table functions

Syntax 1: table function(expression)

- e.g. number of voters and the average cost of voting:

```
subjects.do{
    N = count();
    AvgC = average(C);
}
```

Syntax 2: table function(condition, expression)

- e.g. number of As and the average cost of voting for As:

```
subjects.do{
    Na = count(Type==A);
    AvgCa = average(Type==A, C);
}
```

Table functions

Programs are run sequentially per row

```
subjects.do{
    Type = if( random() < Pa, A, B);
    C = round( random() * MaxC, 1);
    Na = count(Type==A);
    Nb = count(Type==B);
}
```

Incorrect!

Table functions

Programs are run sequentially per row

```
subjects.do{
    Type = if( random() < Pa, A, B);
    C = round( random() * MaxC, 1);
}
subjects.do{
    Na = count(Type==A);
    Nb = count(Type==B);
}
```

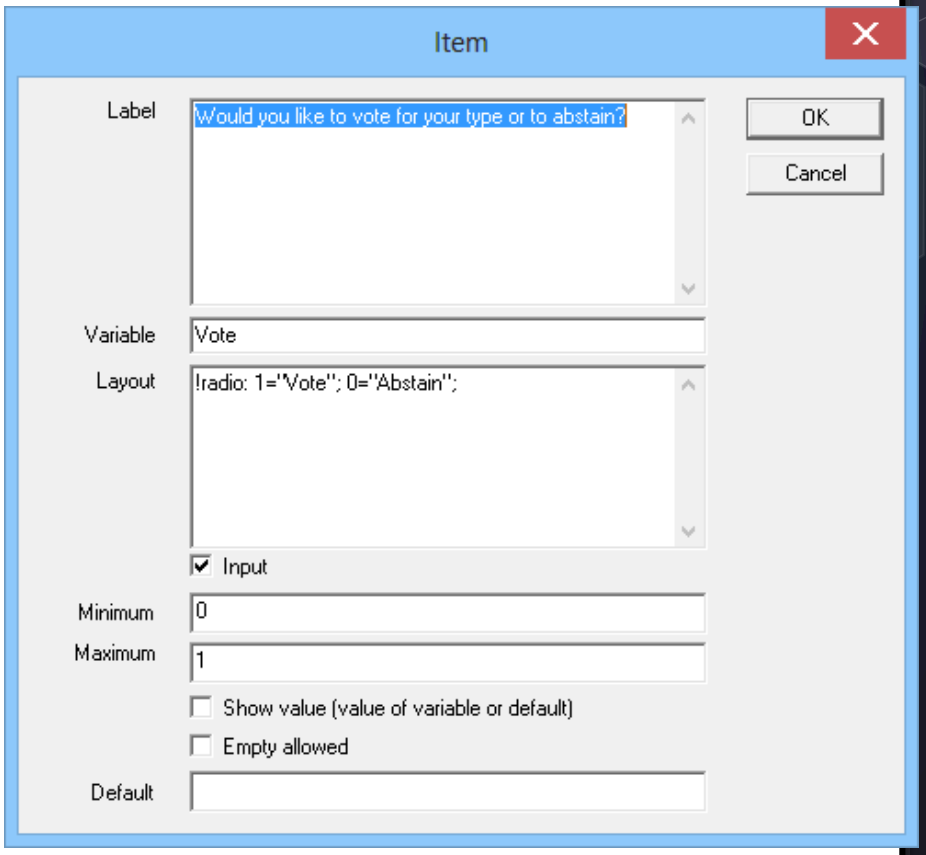
Correct!

Input and output of variables

Items are used for the input and output of variables

- Label (text displayed)
- Variable (for input or output)
- Layout:
  - numbers – radio buttons
  - check boxes – sliders
  - scrollbars – text

  Note
- If the item is used for input we also need a **button**



Input and output of variables

Variables integrated into text

- If instead of displaying “Your type is: A” you want to display “If A wins you earn 100 points but if B wins you earn 0 points” then type the following in the label box

  <>If <Type |!text: A=“A”; B=“B”;> wins you earn 100 points but if <Type |!text: A=“B”; B=“A”;> wins you earn 0 points

RTF is supported so you can do a lot of stuff

- To display “Your *profit* in this period was **–5.00 points**” where the profits are bold only when negative then type

  <>{\rtf Your \i profit \i0 in this period was <Profit |!text: 1=“”; –1=“\b ”;><Profit |0.01> points <Profit |!text: 1=“”; –1=“\b0 ”;>}

globals table

Use the globals table when a variable is the same value for all subjects

```
globals.do{
    Tiebreak = if(random()<0.5, A, B);
}
subjects.do{
    Votesa = count(Vote==1 & Type==A);
    Votesb = count(Vote==1 & Type==B);
    Winner = if(Votesa > Votesb, A, 0) + if(Votesa < Votesb, B, 0)
    + if(Votesa == Votesb, Tiebreak, 0);
    Profit = MaxC + if(Winner == Type, Prize, 0) - C*Vote;
}
```

Groups

In most experiments subjects are divided into groups

Let's redo the rational turnout experiment but with random allocation of voters to groups of 5 and then

- Voters are randomly selected to prefer A or B
  - Probability of preferring a $p_A > ½$
- Voters can vote for A, B, or abstain
  - They get 100 if their preference wins and 0 otherwise
  - Voting is costly: drawn from a uniform distribution $c_i \in [0, 50]$

Groups

The variable Group determines the group matching

- The number of groups can be set in the **background stage**

There are menu commands for different types of matchings (treatment menu):

- Partner
- Stranger
- absolute Stranger
- typed absolute Stranger

Important:

- Before running an experiment, check the **parameter table** (treatment menu)

Groups

The Group variable can also be changed:

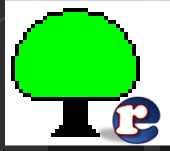
- Manually in the **parameter table**
  - Double-click on each cell and set group
- Through a **program** in the **background stage**, e.g.,

```
subjects.do{
    Group = 1;
    Group = if( Subject >= 6 & Subject <= 15, 2, Group);
    Group = if( Subject > 15, 3, Group);
}
```

Common matching protocols

- **Partners** in groups of size *N*, e.g. N = 4:

```
globals.do{
    N = 4;
}
subjects.do{
    Group = roundup( Subject / N, 1);
}
```



Common matching protocols

- **Strangers** in groups of size *N*, e.g. N = 4:

```
globals.do{
    N = 4;
}
subjects.do{
    RndNum = random();
}
subjects.do{
    Group = roundup( count(RndNum <= :RndNum) / N, 1);
}
```

Common matching protocols

- **Strangers within matching groups** of size *M* and in groups of size *N*, e.g. M = 10 & N = 2:

```
globals.do{
    M = 10;
    N = 2;
}
subjects.do{
    MatchGroup = roundup( Subject / M, 1);
    RndNum = random() + MatchGroup;
}
subjects.do{
    Group = roundup( count(RndNum <= :RndNum) / N, 1);
}
```

same() function

same( ) is the table function used to make group calculations

- e.g. to count the total number of voters, the number of A voters and the number of B voters **within each group**

```
subjects.do{
    N = count( same(Group) );
    Na = count( same(Group) & Type==A);
    Nb = count( same(Group) & Type==B);
}
```

Scope operator

Alternatively, one can use the scope operator ":"

```
subjects.do{
    N = count( Group == :Group );
    Na = count(Group == :Group & Type==A);
    Nb = count(Group == :Group & Type==B);
}
```

Scope operator

Scope operator gives you more flexibility

- e.g. rank voters in the group according to their cost

```
subjects.do{
    RankC = count( same(Group) & C <= C);
}
```

Incorrect!

```
subjects.do{
    RankC = count( same(Group) & C <= :C);
}
```

Correct!

Exp. 2: An ultimatum game

Subjects are matched in pairs

- Each pair has 1 proposer and 1 responder
- Proposers offer responders *x* points from *y* available points
- Responders can accept or reject the offer
  - If the responder accepts:
    - Proposers earn: $\pi_P = y - x$
    - Responders earn: $\pi_R = x$
  - If the responder rejects:
    - Both earn 0 points

Play for *t* periods

- Random matching and random assignment of roles

Assigning types

Player types can be assigned by programming them

- e.g., to randomly allocate one proposer and one responder per pair

```
subjects.do{
    RndNum = random();
}
subjects.do{
    RndOther = find(same(Group) & not( same(Subject) ) ,
    RndNum);
    Proposer = if( RndOther > RndNum, 1, 0);
}
```

Or ... use the parameter table (less flexible)

- period parameters, subject parameters, period × subject parameters

Common matching protocols

- **Typed partners** in groups of size *N* and with *N* types of players where each group has one player of each type and **types are constant** across periods, e.g. N = 2:

```
globals.do{
    N = 2;
}
subjects.do{
    Group = roundup( Subject / N, 1);
    Type = mod(Subject - 1, N) + 1;
}
```

Common matching protocols

- **Typed partners** in groups of size *N* and with *N* types of players where each group has one player of each type and **types are randomly redrawn** every period, e.g. N = 2:

```
globals.do{
    N = 2;
}
subjects.do{
    RndNum = random();
    Group = roundup( Subject / N, 1);
}
subjects.do{
    Type = mod(count(same(Group) & RndNum <= :RndNum) - 1, N) + 1;
}
```

Common matching protocols

- **Typed strangers** in groups of size *N* and with *N* types of players where each group has one player of each type and **types are randomly redrawn** every period, e.g. N = 2:

```
globals.do{
    N = 2;
}
subjects.do{
    RndNum = random();
}
subjects.do{
    Group = roundup( count(RndNum <= :RndNum) / N, 1);
}
subjects.do{
    Type = mod(count(same(Group) & RndNum <= :RndNum) - 1, N) + 1;
}
```

Common matching protocols

- **Typed strangers** in groups of size *N* and with *N* types of players where each group has one player of each type and **types are constant** every period, e.g. N = 2:

```
globals.do{
    N = 2;
    NG = subjects.maximum(Subject) / N;
}
subjects.do{
    Type = mod(Subject - 1, N) + 1;
    RndNum = random();
}
subjects.do{
    Group = mod(count(same(Type) & RndNum <= :RndNum) - 1, NG) +
    1;
}
```

Common matching protocols

- **Typed strangers within matching groups** of size *M* in groups of size *N* and with *N* types of players where each group has one player of each type and **types are randomly redrawn** every period, e.g. M = 10 & N = 2:

```
globals.do{
    M = 10;
    N = 2;
}
subjects.do{
    MatchGroup = roundup( Subject / M, 1);
    RndNum = random() + MatchGroup;
}
subjects.do{
    Group = roundup( count(RndNum <= :RndNum) / N, 1);
}
subjects.do{
    Type = mod(count(same(Group) & RndNum <= :RndNum) - 1, N) + 1;
}
```

Common matching protocols

- **Typed strangers within matching groups** of size *M* in groups of size *N* and with *N* types of players where each group has one player of each type and **types are constant** every period, e.g. M = 10 & N = 2:

```
globals.do{
    M = 10;
    N = 2;
    NG = subjects.maximum(Subject) / N;
}
subjects.do{
    MatchGroup = roundup( Subject / M, 1);
    Type = mod(count(same(MatchGroup) & Subject <= : Subject) - 1, N) + 1;
    RndNum = random() + MatchGroup;
}
subjects.do{
    Group = mod(count(same(Type) & RndNum <= :RndNum) - 1, NG) + 1;
}
```

Sequential vs. simultaneous screens

Rational turnout

Voting decision

Profit display

Ultimatum game

Proposer offer

waiting

waiting

Responder acceptance

Proposer profit display

Responder profit display

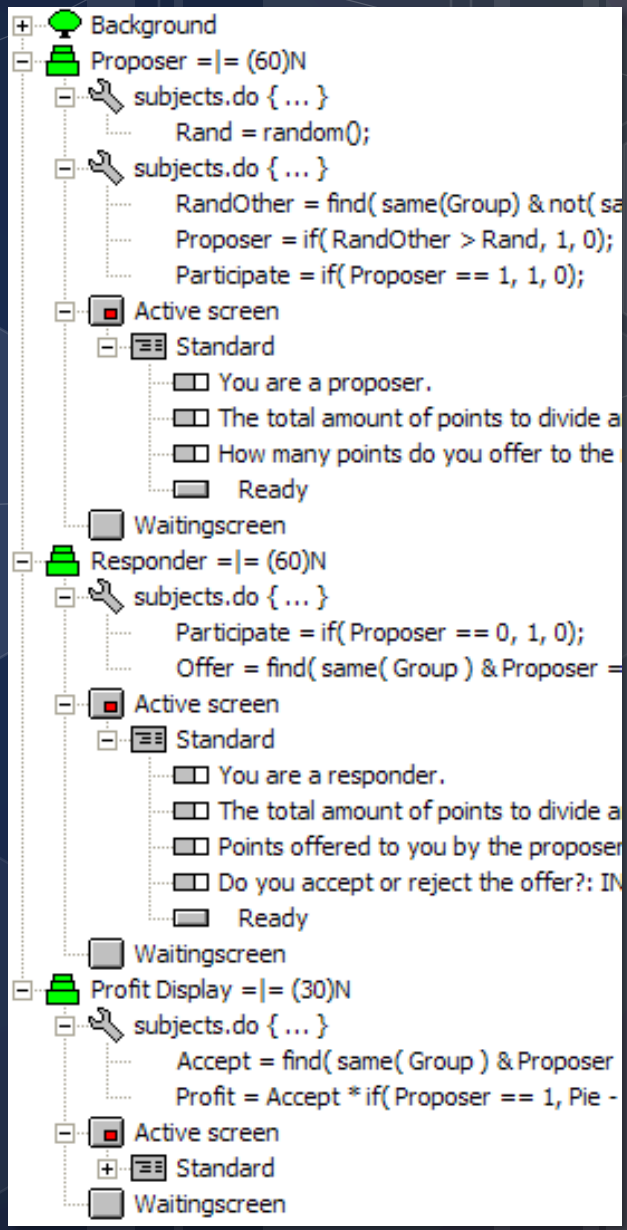
Participate

The variable Participate determines who enters a stage

- Enter stage: Participate = 1.
- Skip stage: Participate = 0.
- At every stage, Participate resets to 1

For the ultimatum game use either

- Participate = if (Proposer == 1, 1, 0);
- Participate = if (Proposer == 0, 1, 0);



Exp. 3: Another ultimatum game

Proposers offer responders *x* points from *y* available points

- Responders state what is the minimum acceptable offer
  - If the offer ≥ minimum acceptable offer:
    - Proposers earn: $\pi_P = y - x$
    - Responders earn: $\pi_R = x$
  - If the offer < minimum acceptable offer:
    - Both earn 0 points

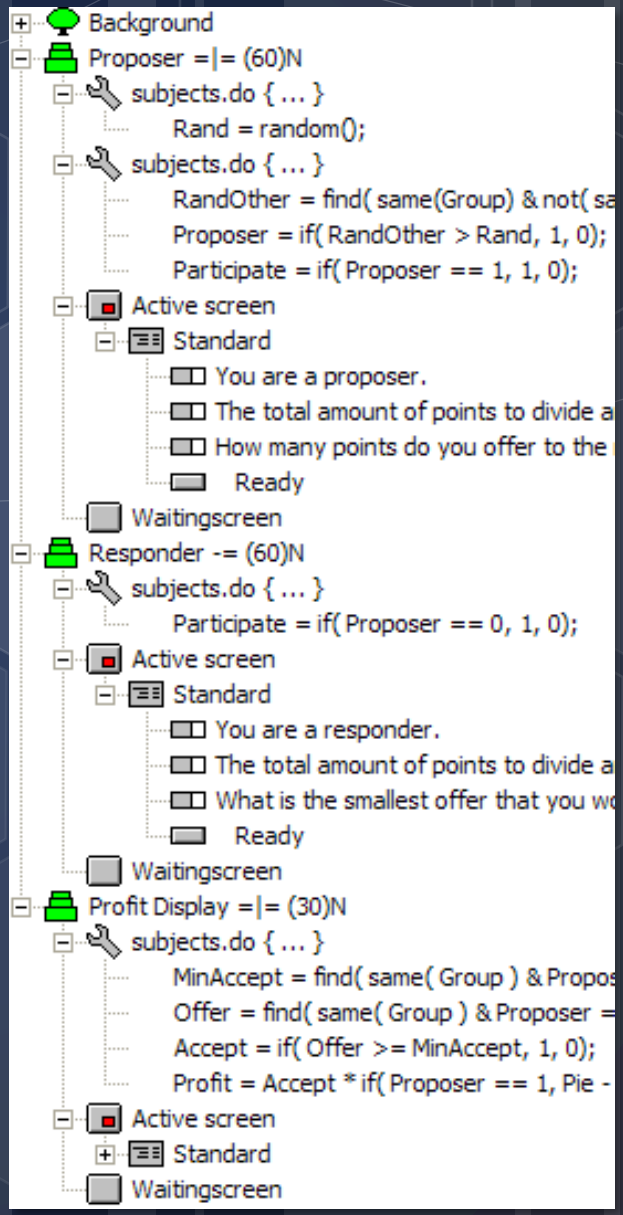
This is an example of the strategy method

Stage: start options

To make proposers and responders decide simultaneously

Stage start property

- Wait for all
  - general case
- Start is possible
  - simultaneous stages
  - stages that do not depend on other participants



Exp. 4: A very simple English auction

Subjects are all buyers

- Subjects get a (random) private value for the auctioned good
- Subjects make bids
- Winner pays the second highest price
- The auction is terminated after a fixed timeout
- Winner gets: $\pi^B = y + v_i - b_2$
- Others get: $\pi^S = y$

For market experiments we need to use the

- **contracts table**
  - new types of boxes: contract creation box, contract list box, and contract grid box

Contracts table

The contracts table has a *flexible* number of records (records can be added)

- New records are created in contract creation boxes
- or with the new command: contracts.new{ x=1; }

Buyer	Bid	Order

Contracts table

The contracts table has a *flexible* number of records (records can be added)

- New records are created in contract creation boxes
- or with the new command: contracts.new{ x=1; }

Buyer	Bid	Order
2	10	1

Subject 2 makes a bid (highest bid)

Contracts table

The contracts table has a *flexible* number of records (records can be added)

- New records are created in contract creation boxes
- or with the new command: contracts.new{ x=1; }

Buyer	Bid	Order	
2	10	2	Subject 2 makes a bid (second highest bid)
5	12	1	Subject 5 makes a bid (highest bid)

Contracts table

The contracts table has a *flexible* number of records (records can be added)

- New records are created in contract creation boxes
- or with the new command: contracts.new{ x=1; }

Buyer	Bid	Order	
2	10	3	Subject 2 makes a bid
5	12	2	Subject 5 makes a bid (second highest bid)
4	15	1	Subject 4 makes a bid (highest bid)

Contracts table

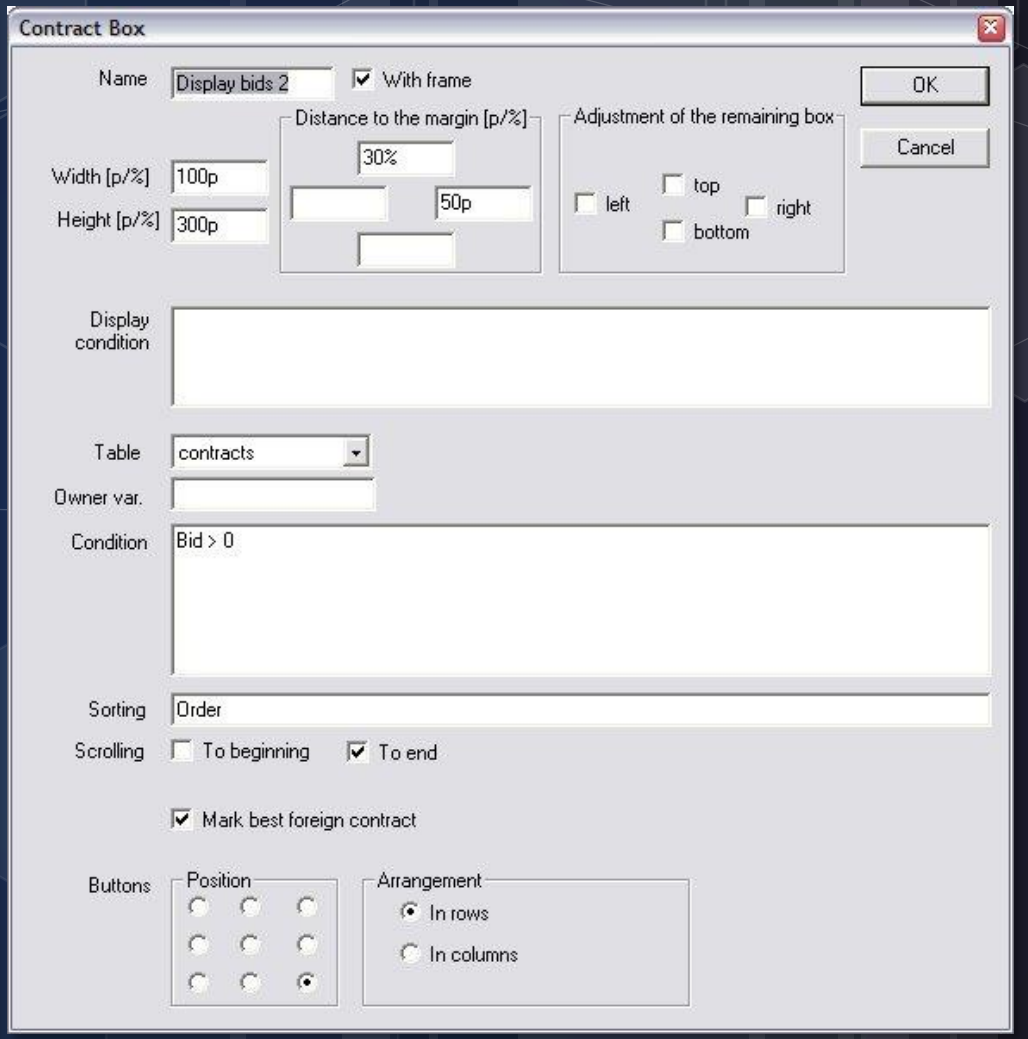
The contracts table has a *flexible* number of records (records can be added)

- New records are created in contract creation boxes
- or with the new command: contracts.new{ x=1; }

Buyer	Bid	Order	
2	10	4	Subject 2 makes a bid
5	12	3	Subject 5 makes a bid
4	15	2	Subject 5 makes a bid (second highest bid)
2	17	1	Subject 2 makes another bid (highest offer)

Contracts table

The contents of the contracts table are displayed with a contracts list box or with a contracts grid box



Exp. 5: A continuous public good game

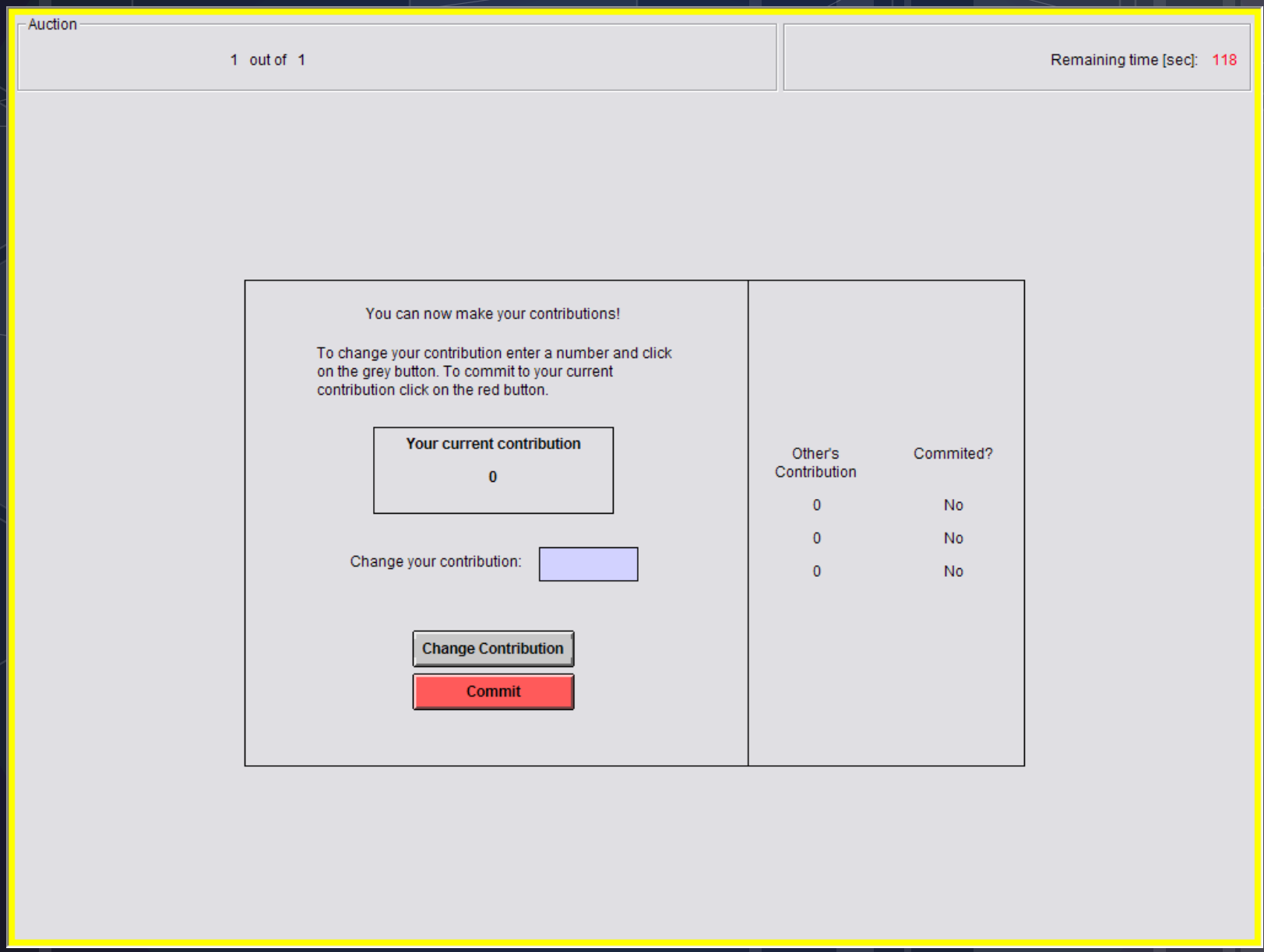
In each period each subject gets 20 points.

- Points can be kept or invested in a public good and each point invested in the public good pays 0.5 to everyone
- The profit of each subject is:

$$\pi_i = 20 - c_i + 0.5 \times \sum_j c_j$$

- There are 90 sec to make non-binding contributions and contributions become binding when the time expires or when the subject chooses to commit him/herself
- Contributions are observed in real-time by everyone

Exp. 5: A continuous public good game



More contracts table

Note that the contracts table can also be used for interaction within the same screen.

- Use the new command to create the table
- Use contract grid boxes
- Important: Changes to variables during the screen are NOT recorded in the data

Other features

Programming

- Loops: while(condition) { statements; }

Complex move structures

- goto next stage if …

Treatments with indefinite length

- end with a given probability
- end when a specific action is taken

Graphics

- Charts
- Display Pictures/Videos

Communication

- Chat box

Questionnaires

Must be run so that the payoff file is written.

Questions with no consequence on payoff.

- Different formats for the questions.
- Layout is not screen oriented: indefinite end with scrollbar.
- Text entry possible.

Typical Questionnaires:

- Address form (writes the payment file)
- Questions concerning their strategies
- Profit display
- Goodbye screen

Planning a simple session

Welcome treatment (welcome.ztt)

- Set the show-up fee
- Control questions

Public goods experiment (pg.ztt)

- The main treatment

Ultimatum game (ug.ztt)

- A second treatment

Questionnaires and payment (end.ztq)

- payment file